第三章 Unix/Linux 多线程编程

[引言]本章在前面章节多线程编程基础知识的基础上,着重介绍 Unix/Linux 系统下的多线程编程接口及编程技术。

3.1 POSIX 的一些基本知识

POSIX 是可移植操作系统接口（Portable Operating System Interface）的首字母缩写。POSIX 是基于 UNIX 的，这一标准意在期望获得源代码级的软件可移植性。换句话说，为一个 POSIX 兼容的操作系统编写的程序，应该可以在任何其它的 POSIX 操作系统（即使是来自另一个厂商）上编译执行。POSIX 标准定义了操作系统应该为应用程序提供的接口：系统调用集。POSIX 是由 IEEE（Institute of Electrical and Electronic Engineering）开发的，并由 ANSI（American National Standards Institute）和 ISO（International Standards Organization）标准化。大多数的操作系统（包括 Windows NT）都倾向于开发它们的变体版本与 POSIX 兼容。

POSIX 现在已经发展成为一个非常庞大的标准族，某些部分正处在开发过程中。表 1-1 给出了 POSIX 标准的几个重要组成部分。POSIX 与 IEEE 1003 和 2003 家族的标准是可互换的。除 1003.1 之外，1003 和 2003 家族也包括在表中。

1003.0	管理 POSIX 开放式系统环境（OSE）。IEEE 在 1995 年通过了这项标准。 ISO 的版本是 ISO/IEC 14252:1996。
1003.1	被广泛接受、用于源代码级别的可移植性标准。1003.1 提供一个操作系统的 C 语言应用编程接口（API）。IEEE 和 ISO 已经在 1990 年通过了这个标准，IEEE 在 1995 年重新修订了该标准。
1003.1b	一个用于实时编程的标准（以前的 P1003.4 或 POSIX.4）。这个标准在 1993 年被 IEEE 通过，被合并进 ISO/IEC 9945-1。
1003.1c	一个用于线程（在一个程序中当前被执行的代码段）的标准。以前是 P1993.4 或 POSIX.4 的一部分，这个标准已经在 1995 年被 IEEE 通过，归入 ISO/IEC 9945-1:1996。
1003.1g	一个关于协议独立接口的标准,该接口可以使一个应用程序通过网络与另一个应用程序通讯。 1996 年，IEEE 通过了这个标准。
1003.2	一个应用于 shell 和工具软件的标准，它们分别是操作系统所必须提供的命令处理器和工具程序。 1992 年 IEEE 通过了这个标准。ISO 也已经通过了这个标准（ISO/IEC 9945-2:1993）。

1003.2d	改进的 1003.2 标准。
1003.5	一个相当于 1003.1 的 Ada 语言的 API。在 1992 年，IEEE 通过了这个标准。并在 1997 年对其进行了修订。ISO 也通过了该标准。
1003.5b	一个相当于 1003.1b（实时扩展）的 Ada 语言的 API。IEEE 和 ISO 都已经通过了这个标准。ISO 的标准是 ISO/IEC 14519:1999。
1003.5c	一个相当于 1003.1q（协议独立接口）的 Ada 语言的 API。在 1998 年， IEEE 通过了这个标准。ISO 也通过了这个标准。
1003.9	一个相当于 1003.1 的 FORTRAN 语言的 API。在 1992 年,IEEE 通过了这个标准，并于 1997 年对其再次确认。ISO 也已经通过了这个标准。
1003.10	一个应用于超级计算应用环境框架（Application Environment Profile，AEP）的标准。在 1995 年，IEEE 通过了这个标准。
1003.13	一个关于应用环境框架的标准，主要针对使用 POSIX 接口的实时应用程序。在 1998 年，IEEE 通过了这个标准。
1003.22	一个针对 POSIX 的关于安全性框架的指南。
1003.23	一个针对用户组织的指南，主要是为了指导用户开发和使用支持操作需求的开放式系统环境（OSE）框架
2003	针对指定和使用是否符合 POSIX 标准的测试方法，有关其定义、一般需求和指导方针的一个标准。在 1997 年，IEEE 通过了这个标准。
2003.1	这个标准规定了针对 1003.1 的 POSIX 测试方法的提供商要提供的一些条件。在 1992 年，IEEE 通过了这个标准。
2003.2	一个定义了被用来检查与 IEEE 1003.2（shell 和 工具 API）是否符合的测试方法的标准。在 1996 年，IEEE 通过了这个标准。

表 3.1 POSIX 标准的重要组成部分

本章将重点讲述“POSIX 线程”，即符合 POSIX 国际正式标准 POSIX1003. 1c-1995 的部分。本章假定用户使用的编程语言为 ANSI C 语言。

3.2 POSIX 线程库

首先，在编写 POSIX 多线程 C 程序时，需要包含头文件’pthread.h’。POSIX 线程函数都以’pthread_’开头。在本章中，我们将介绍一下线程操作函数：

POSIX 函数	描述
pthread_cancel	终止另一个线程
pthread_create	创建一个线程
pthread_detach	设置线程以释放资源
pthread_equal	测试两个线程 ID 是否相等
pthread_exit	退出线程，而不退出进程

pthread_join	等待一个线程
pthread_self	找出自己的线程 ID

表 3.2 POSIX 线程管理函数

3.2.1 创建线程

‘pthread_create’ 函数创建一个线程。

```
int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t* restrict attr,
                   void *(*start_routine)(void *),
                   void *restrict arg);
```

参数 thread 指向保存线程 ID 的 pthread_t 结构。参数 attr 表示一个封装了线程的各种属性的属性对象，用来配置线程的运行，如果为 NULL，则使新线程具有默认的属性。线程属性将在后面的 XX 节讨论。第三个参数 start_routine 是线程开始执行的时候调用的函数的名字。这个函数必须具有以下的格式：

```
void* start_routine(void* arg);
```

返回的 void 指针将被 pthread_join 函数当做退出状态来处理。第四个参数 arg 正是传递给 start_routine 函数的参数。POSIX 的 pthread_create 函数会使创建的线程自动处于可运行状态，而不需要一个单独的启动操作。

如果成功，pthread_create 返回 0，如果不成功，pthread_create 返回一个非零的错误码。下表列出了 pthread_create 的错误形式及相应的错误码

错误	原因
EAGAIN	系统没有创建线程所需的资源，或者创建线程会超出系统对一个进程中线程总数的限制
EINVAL	attr 参数是无效的
EPERM	调用程序没有适当的权限来设定调度策略或 attr 指定的参数

表 3.3 pthread_create 的错误形式及相应的错误码

每一个线程可以通过调用函数 pthread_self 得到本线程的 ID(数据结构类型：pthread_t)，它的形式为：

```
pthread_t pthread_self(void);
```

由于 pthread_t 可能是一个结构，因此 POSIX 提供了一个函数 pthread_equal 来比较线程 ID 是否相等。这个函数的形式为：

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

两个参数 t1 和 t2 是两个线程 ID，如果它们相等，pthread_equal 就返回一个非零值，如果不相等，则返回 0。

3.2.2 分离（Detach）和接合（Join）线程

POSIX 线程的一个特点是：除非线程是被分离了的，否则在线程退出时，它的资源是不会被释放的。pthread_detach 函数用来分离线程：

```
int pthread_detach(pthread_t thread);
```

它设置线程的内部选项来说明线程退出后，其所占有的资源可以被回收。参数 thread 是要分离的线程的 ID。被分离的的线程退出时不会报告它们的状态。如果函数调用成功，pthread_detach 返回 0，如果不成功，pthread_detach 返回一个非零的错误码。下表列出了 pthread_detach 的错误形式及相应的错误码

错误	原因
EINVAL	thread 对应的不是一个可分离的线程.
ESRCH	没有 ID 为 thread 的线程

表 3.4 ‘pthread_detach’的错误形式及相应的错误码

pthread_join 函数可以使调用这个函数的线程等待指定的线程运行完成再继续执行。它的形式为：

```
int pthread_join(pthread_t thread, void **value_ptr);
```

参数 thread 为要等待的线程的 ID，参数 value_ptr 为指向返回值的指针提供一个位置，这个返回值是由目标线程传递给 pthread_exit 或 return 的。如果 value_ptr 为 NULL，调用程序就不会对目标线程的返回状态进行检索了。如果函数调用成功，pthread_join 返回 0，如果不成功，pthread_join 返回一个非零的错误码。下表列出了 pthread_join 的错误形式及相应的错误码

错误	原因
EINVAL	thread 对应的不是一个可接合的线程
ESRCH	没有 ID 为 thread 的线程

表 3.5 pthread_join 的错误形式及相应的错误码

如果线程没有被分离，并且执行 pthread_join(pthread_self())，那么该线程将被一直挂起，因为这条语句造成了死锁。有些 POSIX 的实现可以检测到死锁，并迫使 pthread_join 带着错误 EDEADLK 返回，但是，POSIX 并不要求一定要进行这种检测。

3.2.3 退出和取消线程

进程的终止可以通过在主函数 main()中直接调用 exit、return、或者通过进程中的任何其它线程调用 exit 来实现。在任何一种情况下，该进程的所有线程都会终止。如果主线程在创建了其它线程之后没有工作可做，它就应该阻塞到所有线程都结束为止，或者应该调用 pthread_exit(NULL)。

有时程序不必等待线程执行完成，这时程序需要使线程中途退出。POSIX 线程库提供了两个撤销线程的函数 pthread_exit 和 pthread_cancel。下面对这两个函数分别进行介绍。

pthread_exit 函数可以使调用这个函数的线程中止运行，并且允许线程传递一个指针，这个指针可以用来指向线程的返回值。它的形式为：

```
void pthread_exit(void *value_ptr);
```

连接了这个线程可以获得参数 value_ptr 的值。回顾前面介绍的 pthread_join 函数，这个函数的参数 void **value_ptr，正是保存 pthread_exit 函数的参数 void *value_ptr 的地址。这里要注意，pthread_exit 的参数 value_ptr 必须指向线程退出后仍然存在的数据。

POSIX 没有为 pthread_exit 定义任何错误。

POSIX doesn't define any error code for ‘pthread_exit’.

线程也可以通过取消机制迫使其它的线程退出。线程可以调用函数 pthread_cancel 来请求取消另一个线程。这个函数的形式是：

```
int pthread_cancel(pthread_t thread);
```

参数 thread 是要取消的目标线程的线程 ID。该函数并不阻塞调用线程，它发出取消请求后就返回了。如果成功，pthread_cancel 返回 0，如果不成功，pthread_cancel 返回一个非零的错误码。

线程收到一个取消请求时会发生什么情况取决于它的状态和类型。如果线程处于 PTHREAD_CANCEL_ENABLE 状态，它就接受取消请求，如果线程处于 PTHREAD_CANCEL_DISABLE 状态，取消请求就会被保持在挂起状态。默认情况下，线程处于 PTHREAD_CANCEL_ENABLE 状态。

pthread_setcancelstate 函数用来改变调用线程的取消状态，它的形式为：

```
int pthread_setcancelstate(int state, int *oldstate);
```

参数 state 表示要设置的新状态，参数 oldstate 为一个指向整形的指针，用于保存线程以前的状态。如果成功，该函数返回 0，如果不成功，它返回一个非 0 的错误码。通常情况下，线程函数在改变了线程的取消状态之后，应该在执行完某些操作之后恢复线程的取消状态，

否则，对于其它可能取消该线程的线程而言，取消操作的结果将无法预测，这很可能不利于程序的正确执行。

当线程将退出作为对取消请求的响应时，取消类型允许线程控制它在什么地方退出。当它的取消类型为 PTHREAD_CANCEL_ASYNCHRONOUS 时，线程在任何时候都可以响应取消请求。当它的取消类型为 PTHREAD_CANCEL_DEFERRED 时，线程只能在特定的几个取消点上响应取消请求。在默认情况下，线程的类型为 PTHREAD_CANCEL_DEFERRED。

pthread_setcanceltype 函数用来修改线程的取消类型。它的形式为：

```
int pthread_setcanceltype(int type, int *oldtype);
```

参数 type 指定线程的取消类型，参数 oldtype 用来指定保存原来的取消类型的地址。如果成功，该函数返回 0，如果不成功，它返回一个非 0 的错误码。

线程可以通过调用 pthread_testcancel 在代码中的特定的位置上设置一个取消点。当类型为 PTHREAD_CANCEL_DEFERRED 的线程到达这样一个取消点时，就接受挂起的取消请求。该函数的形式为:

```
void pthread_testcancel(void);
```

3.2.4 用户级线程与内核级线程

用户级线程(user-level thread)和内核级线程(kernel-level thread)是两种传统的线程控制模式。用户级线程通常都运行在一个现存的操作系统之上。这些线程对内核来说是不可见的，它们被封装在进程里，并竞争分配给进程的资源。线程由一个线程运行系统来调度，这个系统是进程代码的一部分。带有用户级线程的程序通常会连接到一个特殊的库上去，这个库中的每个库函数都用外套(jacket)包装起来。在调用被外套包装的库函数之前，外套函数要调用线程运行系统来进行线程管理，在调用了被外套包装的库函数之后，外套函数可能也要进行这样的操作。

这样做的必要性是为了解决下面的情况：由于 read 或 sleep 这样的函数可能会使进程阻塞，所以它们给用户级线程带来了一个问题，那就是要避免某个线程在调用这些阻塞型函数之后，整个进程被阻塞。这就要求用户级线程库用一个无阻塞的版本来替换每一个外套包装的、潜在的阻塞型调用。线程运行系统通过测试来查看调用是否会使线程阻塞，如果调用不会阻塞，运行系统就立即进行调用，但是，如果调用会阻塞，运行系统就会将线程放在一个等待线程的列表中，将调用添加到一个动作列表中，以便稍后再试，然后挑选另一个线程来运行。所有这些控制过程对用户和操作系统来说都是不可见的。

用户级线程的开销很低，但是它们也有些缺点。用户线程模型假定线程运行系统最终会重新获得控制权，这可能会受到 CPU 绑定线程(CPU-bound thread)的阻碍。CPU 绑定线程很少执行库函数调用，这样就会阻止线程运行系统重新获得控制权来调度其它的线程。程序员必须要显式地迫使 CPU 绑定线程在适当的地方放弃对 CPU 的控制，以避免出现封锁状态。第二个问题是，用户级线程只能共享分配给它们的封装进程的处理器资源。因为线程一次只能运行

在一个处理器上，这种约束限制了可用的并行总量。使用线程的主要原因之一就是要利用多处理器工作站的优势，所以仅使用用户级线程本身并不是一种能让人接受的方法。

对内核级线程来说，内核了解每一个作为可调度实体的线程，这些线程可以在全系统范围内竞争处理器资源。内核级线程的调度开销可能和进程自身的调度差不多昂贵，但是，内核级线程可以利用多处理器的优势。内核级线程的同步和数据共享比整个进程的同步和数据共享的开销要低一些，但内核级线程的管理比用户级线程的管理代价更高。

还有一种模型，称作混合线程模型(hybrid thread model)，它通过提供两个级别的控制，同时具备了用户级和内核级模型的优点。用户用用户级线程编写程序，然后说明有多少个内核可调度实体与这个进程相关。运行时，将用户级线程映射为系统的可调度实体，以实现并行。用户拥有的映射控制级别取决于实现，例如在 Sun 的 Solaris 线程实现中，用户级线程被称为线程，而内核可调度实体被称为轻量级进程(lightweight process)。用户可以指定由一个特定的轻量级进程来运行制定的线程，或者由一个轻量级进程池来运行一组制定的线程。

POSIX 线程调度模型是一个混合模型，它很灵活，足以在标准的特定实现中支持用户级和内核级的线程。模型中包括两级调度——线程级和内核实体级。线程与用户级线程类似，内核实体由内核调度。由线程库来决定它需要多少内核实体，以及它们是如何映射的。

POSIX 引入了一个线程调度竞争范围(thread-scheduling contention scope)的概念，这个概念赋予了程序员一些控制权，使它们可以控制怎样将内核实体映射为线程。线程的 contentionscope 属性可以是 PTHREAD_SCOPE_PROCESS，也可以是 PTHREAD_SCOPE_SYSTEM。带有 PTHREAD_SCOPE_PROCESS 属性的线程与它们所在的进程中的其它线程竞争处理器资源。POSIX 没有说明这样一个线程怎样与它所在的进程中的其它线程竞争，因此 PTHREAD_SCOPE_PROCESS 线程可以是严格的用户级线程，或者它们也可以使用某种更复杂的方式映射到一个内核实体池中去。带有 PTHREAD_SCOPE_SYSTEM 属性的线程很像内核级线程，他们在全系统范围内竞争处理器资源。POSIX 将 PTHREAD_SCOPE_SYSTEM 线程和内核实体之间的映射留给具体实现来完成，但是一种明显的映射方式是，将这样一个线程直接与内核实体绑定起来。POSIX 线程的具体实现可能支持 PTHREAD_SCOPE_PROCESS、或 PTHREAD_SCOPE_SYSTEM 或者两者都支持。

3.2.5 线程的属性

POSIX 将栈的大小和调度策略这样的特征封装到一个 pthread_attr_t 类型的对象中去，用面向对象的方式表示和设置特征。属性对象只在线程创建的时候会对线程产生影响。编写程序时可以先创建一个属性对象，然后再将栈的大小和调度策略这样的特征与属性对象关联起来，之后就可以通过向 pthread_create 传递相同的线程属性对象来创建多个具有相同特征的线程。通过将各种特征组合到单个对象中去，POSIX 避免了用大量参数来调用 pthread_create 的情况。

表 3.6 显示的是线程属性的可设置特征及其相关函数，后面我们将对这些特征和函数进行讨论。

特征	函数
属性对象	pthread_attr_destroy pthread_attr_init
状态	pthread_attr_getdetachstate pthread_attr_setdetachstate
栈	pthread_attr_getguardsize pthread_attr_setguardsize pthread_attr_getstack pthread_attr_setstack
调度	pthread_attr_getinheritsched pthread_attr_setinheritsched pthread_attr_getschedparam pthread_attr_setschedparam pthread_attr_getschedpolicy pthread_attr_setschedpolicy pthread_attr_getscope pthread_attr_setscope

表 3.6 线程属性的可设置特征及其相关函数

函数 pthread_attr_init 用默认值对一个线程属性对象进行初始化。pthread_attr_destroy 函数将属性对象的值设为无效的。被设为无效的属性对象可以再次被初始化为一个新的属性对象。pthread_attr_init 和 pthread_attr_destroy 都只有一个参数，即一个指向属性对象的指针。这两个函数的形式为：

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

如果成功，函数返回 0，如果不成功，函数返回一个非 0 的错误码。

大多数针对属性对象的函数都是获取或设置属性对象的属性。第一个参数是一个指向属性对象的指针。对于获取操作，第二个参数是一个指向存放值的位置的指针，而对于设置操作，第二个参数是属性的设置值。因此，后面读者可以根据函数参数的名称和类型推断出参数的含义，我们就不一一介绍了。

3.2.5.1 线程状态

线程状态的可能取值为 PTHREAD_CREATE_JOINABLE 和 PTHREAD_CREATE_DETACHED。pthread_attr_getdetachstate 函数用来查看一个属性对象中的线程状态，而 pthread_attr_setdetachstate 函数用来设置一个属性对象中的线程状态。这两个函数的形式为：

```
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

如果成功，这些函数都返回 0。如果不成功，他们就返回一个非零的错误码。

如前所述，可以通过调用 pthread_detach 函数来分离一个线程，现在也可以通过先设置属性对象的线程状态为 PTHREAD_CREATE_DETACHED，并在创建线程时传递这个属性对象，使线程处于分离状态。被分离的线程是不能用 pthread_join 来等待的。默认情况下，线程是可接合的。

3.2.5.2 线程栈

线程有自己的栈，用户可以设置这个栈的位置和大小，这就必须先用特定的栈属性来创建一个属性对象，然后把这个属性对象传递给 pthread_create 来创建线程。

pthread_attr_getstack 函数用来查看栈的参数，pthread_attr_setstack 函数用来设置一个属性对象的栈参数。这两个函数的形式为：

```
int pthread_attr_getstack(const pthread_attr_t *restrict attr, void **restrict stackaddr, size_t *restrict stacksize);
int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr, size_t stacksize);
```

如果成功，这些函数都返回 0。如果不成功，他们就返回一个非零的错误码。如果 stacksize 超出了范围，pthread_attr_setstack 函数就将 errno 设置为 EINVAL。

如果没有为线程指定堆栈，用户可以调用 POSIX 提供的检查栈溢出或者为栈溢出设置警戒的函数。pthread_attr_getguardsize 函数用来查看警戒参数，pthread_attr_setguardsize 函数在一个属性对象中设置了用来控制栈溢出的警戒参数。如果参数 guardsize 为 0，栈就是无警戒的，如果非 0，那么线程栈将至少多获得 guardsize 的额外内存。对这个额外内存区的溢出会引发一个错误。这两个函数的形式为：

```
int pthread_attr_getguardsize(const pthread_attr_t * restrict attr, size_t *restrict guardsize);
int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
```

如果成功，这些函数都返回 0。如果不成功，他们就返回一个非零的错误码。如果参数 attr 或 guardsize 是无效的，他们就返回 EINVAL。

POSIX 规定线程的栈如果被应用程序自己定义的话，栈的空间管理(大小确定，空间伸缩等)需要应用程序自己管理，比如应用程序定义了自己的栈大小是 10M，应用程序必须确定这 10M 空间够大，而且在空间不够的情况下，应用程序需要自己扩展栈的大小，否则的话，可能会发生栈溢出的错误。如果应用程序不自定义栈的话，POSIX 的线程实现机制会保证应用程序的线程栈按照运行情况自动调整。除了主线程以外，其他的线程的栈的管理都是在堆里实现的。原则上不建议应用程序管理线程栈，除非应用程序在空间上需要精心设计。

3.2.5.3 线程调度

线程调度的竞争范围控制了线程是在进程内部还是在系统级竞争调度资源。pthread_attr_getscope 用来查看竞争范围，pthread_attr_setscope 用来设置一个属性对象的竞争范围。这两个函数的形式为：

```
int pthread_attr_getscope(const pthread_attr_t *restrict attr, int *restrict contentionscope);
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

参数 contentionscope 的可能取值为 PTHREAD_SCOPE_PROCESS 和 PTHREAD_SCOPE_SYSTEM。如果成功，这些函数都返回 0。如果不成功，他们就返回一个非零的错误码。

POSIX 允许线程用不同的方式继承调度策略。pthread_attr_getinheritsched 函数用于查看调度继承策略，而 pthread_attr_setinheritsched 用于为一个属性对象设置调度继承策略。这两个函数的形式为：

```
int pthread_attr_getinheritsched(const pthread_attr_t *restrict attr, int *restrict inheritsched);
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);
```

inheritsched 有两个可能的取值：PTHREAD_INHERIT_SCHED 和 PTHREAD_EXPLICIT_SCHED。使用 PTHREAD_INHERIT_SCHED 时，调度属性从创建线程中继承，传递的属性对象中的调度属性将被忽略。使用 PTHREAD_EXPLICIT_SCHED 时，线程使用属性对象中的调度属性。如果成功，这些函数都返回 0。如果不成功，他们就返回一个非零的错误码。

pthread_attr_getschedpolicy 函数负责获取调度策略，pthread_attr_setschedpolicy 函数负责设置属性对象的调度策略。这两个函数的形式为：

```
int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr, int *restrict policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

参数 policy 为调度策略。头文件 sched.h 为先进先出调度策略定义了 SCHED_FIFO，为轮转调度定义了 SCHED_RR，并为一些其他的策略定义了 SCHED_OTHER。实现 POSIX 线程库的操作系统也可以有自己的调度策略。如果成功，这些函数都返回 0。如果不成功，他们就返回一个非零的错误码。

pthread_attr_getschedparam 函数负责查看调度参数，而 pthread_attr_setschedparam 负责设置一个属性对象的调度参数。这两个函数的形式为：

```
int pthread_attr_getschedparam(const pthread_attr_t *restrict attr, struct sched_param *restrict param);
int pthread_attr_setschedparam(pthread_attr_t *restrict attr, const struct
```

sched_param *restrict param);

如果成功，这些函数都返回 0。如果不成功，他们就返回一个非零的错误码。

参数 struct sched_param 是一个定义在 sched.h 头文件中的结构，它为特定的调度策略服务。SCHED_FIFO 和 SCHED_RR 调度策略只使用这个结构中的 sched_priority 成员。sched_priority 成员为一个 int 型的优先级值，较大的优先级值对应于较高的优先级。实现 POSIX 线程库的操作系统必须至少支持 32 个优先级。

3.2.6 线程安全函数

线程中隐藏的一个问题是它们可能会调用非线程安全的库函数，这样可能会产生错误的结果。如果多个线程能够同时执行函数的多个活动请求而不会相互干扰，那么这个函数就是线程安全的(thread-safe)。POSIX 规定，除了表 3.7 列出的特定的函数之外，所有必需的函数，包括来自标准 C 库中的函数，都要用线程安全的方式来实现。有些函数的传统接口会妨碍它们成为线程安全的函数，这些函数一定要有一个以后缀_r 表示对应的线程安全版本。

有些函数不一定非要是线程安全的，strerror 就是这种函数的一个很重要的例子。尽管不能保证 strerror 是线程安全的，但是很多系统都用线程安全的模式来实现这个函数。不幸的是，由于 strerror 被列在表 3.7 中，所以如果有多个线程调用它的话，你就不能假设它能正确地工作。一般我们只在主线程中使用 strerror，通常为 pthread_create 和 pthread_join 产生的错误消息。而各个子线程也可能产生自己的错误，这时调用 strerror 就可能产生线程不安全问题，因为在传统的 UNIX 实现中，errno 是一个全局外部变量，当系统函数产生一个错误时就会设置 errno。对多线程来说，这种实现方式是无法工作的，因为每个线程都应该有自己的 errno。有些系统实现了线程安全的 strerror_r 版本，而对于那些没有实现 strerror_r 的系统，也可以自己编写线程安全的 strerror_r 函数。

asctime	fcvt	getpwnam	nl-langinfo
basename	ftw	getpwuid	ptsname
catgets	gcvt	getservbyname	putc_unlocked
crypt	getc_unlocked	getservbyport	putchar_unlocked
ctime	getchar-unlocked	getservent	putenv
dbm_clearerr	getdate	getutxent	pututxline
dbm_close	getenv	getutxid	rand
dbm_delete	getgrent	getutxline	readdir
dbm_error	getgrid	gmtime	setenv
dbm_fetch	getgrnam	hcreate	setgrent
dbm_firstkey	gethostbyaddr	hdestroy	setkey
dbm_nextkey	gethostbyname	hsearch	setpwent
dbm_open	gethostent	inet_ntoa	setutxent
dbm_store	getlogin	l64a	strerror
dirname	getnetbyaddr	lgamma	strtok
dlerror	getnetbyname	lgammaf	ttyname

drand48	getnetent	lgammal	unsetenv
ecvt	getopt	localeconv	wcstombs
encrypt	getprotobyname	localtime	wctomb
endgrent	getprotobynumber	lrand48	
endpwent	getprotoent	mrand48	
endutxent	getpwent	nftw	

表 3.7 POSIX 规定的非线程安全的函数

表 3.7 列出了 POSIX 规定的非线程安全的函数，除了该表中的函数，其它的函数必须要用线程安全的方式实现。

虽然表 3.7 中的函数不是必须线程安全的,但某些系统仍然实现了其中一些函数的线程安全版本，比如 asctime 函数对应的线程安全版本 asctime_r，getgrid 函数对应的线程安全版本 getgrid_r 等。

3.2.7 线程特定数据

在单线程程序中，函数经常使用全局变量或静态变量，这是不会影响程序的正确性的，但如果线程调用的函数使用全局变量或静态变量，则很可能引起编程错误，因为这些函数使用的全局变量和静态变量无法为不同的线程保存各自的值,而当同一进程内的不同线程几乎同时调用这样的函数时就可能会有问题发生。而解决这一问题的一种方式就是使用线程特定数据的机制。

下面我们引入一个简单程序实例，并以此作为介绍线程特定数据的案例。

代码 3. 1 线程特定数据

```
static char str[100];
    void A(char* s)
    {
        strncpy(str, s, 100);
    }
    void B()
    {
        printf(“%s\n”, str);
    }
```

可以想象，如果在多线程程序中，各个线程都依次调用函数 A 和函数 B，那么某些线程可能得不到期望的显示结果，因为它使用 B 显示的字符串可能并不是在 A 中设置的字符串。读者会发现，这两个函数非常的简单，但在本章内容中，这两个函数已经足以解释线程特定数据的含义，因为这两个函数代表了使用线程特定数据机制的一种典型场合，即有多个函数使用同一个全局变量。

POSIX 要求实现 POSIX 的系统为每个进程维护一个称之为 Key 的结构数组，这个数组中的每

一个结构称之为一个线程特定数据元素。POSIX 规定系统实现的 Key 结构数组必须包含不少于 128 个线程特定数据元素，而每个线程特定数据元素中至少包含两项内容：使用标志和析构函数指针。线程特定数据元素中的使用标志指示这个数组元素是否正在使用，初始值为“不在使用”，我们稍后讨论线程特定数据元素中的析构函数指针。在后面的介绍中，我们假设 Key 结构数组中包含 128 个元素。

Key 结构数组中每个元素的索引（0~127）称之为键（key），当一个线程调用 pthread_key_create 创建一个新的线程特定数据元素时，系统搜索其所在进程的 Key 结构数组，找出其中第一个不在使用的元素，并返回该元素的键。这个函数的形式为：

```
int pthread_key_create(pthread_key_t *keyptr, void (* destructor)(void *value));
```

参数 keyptr 为一个 pthread_key_t 变量的指针，用于保存得到的键值。参数 destructor 为指定的析构函数的指针。

除了 Key 结构数组，系统还在进程中维护关于每个线程的多种信息。这些特定于线程的信息被保存于称之为 Pthread 的结构中。Pthread 结构中包含名为 pkey 的指针数组，其长度为 128，初始值为空。这 128 个指针与 Key 结构数组的 128 个线程特定数据元素一一对应。在调用 pthread_key_create 得到一个键之后，每个线程可以依据这个键操作自己的 pkey 指针数组中对应的指针，这通过 pthread_getspecific 和 pthread_setspecific 函数来实现。这两个函数的形式为：

```
void *pthread_getspecific(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void *value);
```

pthread_getspecific 返回 pkey 中对应于 key 的指针，而 pthread_setspecific 将 pkey 中对应于 key 的指针设置为 value。

我们使用线程特定数据机制，就是要使线程中的函数可以共享一些数据。如果我们在线程中通过 malloc 获得一块内存，并把这块内存的指针通过 pthread_setspecific 设置到 pkey 指针数组中对应于 key 的位置，那么线程中调用的函数即可通过 pthread_getspecific 获得这个指针，这就实现了线程内部数据在各个函数间的共享。当一个线程终止时，系统将扫描该线程的 pkey 数组，为每个非空的 pkey 指针调用相应的析构函数，因此只要将执行回收的函数的指针在调用 pthread_key_create 时作为函数的参数，即可在线程终止时自动回收分配的内存区。

下面我们可以通过实例来理解线程特定数据的机制：

代码 3. 2 线程特定数据的机制

```
#include  ‘stdio.h’
#include  ‘pthread.h’
#include  ‘string.h’
```

```
#define LEN 100
pthread_key_t key;

void A(char *s)
{
    char *str = (char*)pthread_getspecific(key);
    strncpy(str, s, LEN);
}

void B()
{
    char *str = (char*)pthread_getspecific(key)
    printf("%s\n", str);
}

void destructor(void *ptr)
{
    free(ptr);
    printf("memory freed\n");
}

void *threadfunc1(void *pvoid)
{
    pthread_setspecific(key, malloc(LEN));
    A("Thread1");
    B();
}

void *threadfunc2(void *pvoid)
{
    pthread_setspecific(key, malloc(LEN));
    A("Thread2");
    B();
}
int main()
{
    pthread_t tid1, tid2;

    pthread_key_create(&key, destructor);

    pthread_create(&tid1,  NULL,  &threadfunc1,
NULL);
    pthread_create(&tid2,  NULL,  &threadfunc2,
NULL);
```

```
    pthread_exit(NULL);
    return 0;
}
```

在这个程序中，函数 A 和函数 B 共享了一个内存区，而这个内存区是特定于调用 A 和 B 的线程的，对于其它线程，这个内存区是不可见的。这就安全有效地达到了在线程中的各个函数之间共享数据的目的。

3.2.8 一个 POSIX 多线程实例

代码 3.3 一个 POSIX 多线程实例

```
#include  ‘stdio.h’
#include  ‘pthread.h’

void *threadfunc(void *pvoid)
{
    int id = (int)pvoid;
    printf("Child                    thread%d
says:Hello!World!\n", id);

    return NULL;
}

int main()
{
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, &threadfunc,
1);
    pthread_create(&tid2, NULL, &threadfunc,
2);

    pthread_detach(tid1);
    pthread_join(tid2, NULL);
    printf("Main                       thread
says:Hello!World!\n");

    return 0;
}
```

这是一个十分简单的 POSIX 多线程程序。主线程创建两个子线程，并分别将 1 和 2 作为参数

传递给子线程，之后主线程将第一个子线程分离，将第二个子线程接合，之后主线程显示 Hello! World!。两个子线程则执行相同的线程函数，先将参数 pvoid 转换成 int 型，然后再显示 Hello! World!，其中加入了创建线程时传递的线程编号。

读者可以尝试将 pthread_join 语句去掉，或者换成 pthread_exit(NULL)，看看会是怎样的效果。

3.3 线程通信

3.3.1 互斥量

互斥量是一种特殊的变量，它可以处于锁定状态(locked)，也可以处于解锁状态(unlocked)状态。如果互斥量是锁定的，那么必然有一个线程持有或拥有这个互斥量。如果没有任何一个线程持有这个互斥量，那么这个互斥量就处于解锁、空闲或可用状态。当互斥量空闲，并且有一个线程试图获取这个互斥量时，这个线程就可以获得这个互斥量而不会被阻塞。如果互斥量处于锁定状态，那么试图获取这个互斥量的线程将被阻塞，并加入到这个互斥量的等待队列中。等待队列中的线程获得互斥量的顺序由实现系统决定。这样的机制解决了共享资源的互斥（Mutual Exclusive）访问问题。

3.3.1.1 创建并初始化一个互斥量

POSIX 使用 pthread_mutex_t 类型的变量来表示互斥量。程序在使用 pthread_mutex_t 变量之前，必须对其进行初始化。对于静态分配的 pthread_mutex_t 变量，只要将 PTHREAD_MUTEX_INITIALIZER 赋给这个变量即可，对于动态创建或不使用默认属性的互斥量来说，就要调用 pthread_mutex_init 函数来对其进行初始化。pthread_mutex_init 函数的形式为：

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
```

参数 mutex 是指向要初始化的互斥量的指针，参数 attr 是设定互斥量属性的变量的指针，如果为 NULL，则使互斥量拥有默认属性。

如果成功，pthread_mutex_init 返回 0，如果不成功，pthread_mutex_init 返回一个非零的错误码。下表列出了 pthread_mutex_init 定义的错误码和错误原因：

错误	原因
EAGAIN	系统缺乏初始化*mutex 所需的非内存资源
ENOMEM	系统缺乏初始化*mutex 所需的内存资源
EPERM	调用程序没有适当的优先级

表 3.8 pthread_mutex_init 定义的错误码和错误原因

静态初始化通常比 pthread_mutex_init 更有效，而且可以在定义为全局变量时即完成初始化，这样可以保证在任何线程开始执行之前，初始化即已完成。

6.3.1.2 销毁一个互斥量

当不再使用已经定义了的互斥量时，需要将互斥量销毁。函数 pthread_mutex_destory 用于销毁互斥量。它的形式为：

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

参数 mutex 指向要销毁的互斥量。如果成功，函数返回 0，如果不成功，函数返回一个非零的错误码。

可以用 pthread_mutex_init 重新初始化被销毁的互斥量。

3.3.1.3 对互斥量的锁定和解锁

POSIX 中有两个可以用来获取互斥量的函数，pthread_mutex_lock 和 pthread_mutex_trylock。pthread_mutex_lock 函数会使调用这个函数的线程一直阻塞到互斥量可用为止，而 pthread_mutex_trylock 会立即返回，如果互斥量空闲，那么调用这个函数的线程将获得互斥量，否则函数将返回 EBUSY。pthread_mutex_unlock 用于释放互斥量。这三个函数的形式为：

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

如果成功，这些函数就返回 0，如果不成功，这些函数就返回一个非零的错误码。

3.3.1.4 互斥量应用实例

由于在大多数机器中，对变量的增量和减量操作都不是原子的。比如通常情况下，增量操作包括三个步骤：将内存中的数值装载到 CPU 寄存器中，将寄存器的值加 1，将寄存器中的值写回内存。而机器并不保证在这三步之间不会发生调度，这就导致对变量的增量和减量操作有可能得不到期望的结果。比如，当某线程正在进行增量操作，寄存器中的数值被加 1 之后发生了线程切换，另一个线程对同一个变量进行了增量操作后又切换回原线程，原线程将寄存器中的值写回内存。此时，变量只被增加了 1，而不是增加了 2。

要解决这个问题，就要保证对变量的增量和减量操作是原子的，也就是说，在操作过程中不允许发生线程切换。读者可以在程序中定义如下两个函数，使用它们来进行变量的增量和减量操作。

代码 3.4 互斥量应用实例

```
int increase(int *integer)
```

```
{
    static pthreaad_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
    int error;
    if(error = pthread_mutex_lock(&lock);
        return error;
    *integer++;
    return pthread_mutex_unlock(&lock);
}

int decrease(int *integer)
{
    static pthreaad_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
    int error;
    if(error = pthread_mutex_lock(&lock);
        return error;
    *integer--;
    return pthread_mutex_unlock(&lock);
}
```

3.3.2 条件变量

条件变量是用来通知共享数据的状态信息的机制。由于涉及共享数据，因此条件变量是结合互斥量来使用的。

3.3.2.1 创建和销毁条件变量

POSIX 用 pthread_cond_t 类型的变量来表示条件变量。程序必须在使用 pthread_cond_t 变量之前对其进行初始化。对那些静态分配的、使用默认属性的 pthread_cond_t 变量来说，可以直接将 PTHREAD_COND_INITIALIZER 赋给变量就可以完成初始化。对那些动态分配的或不使用默认属性的变量来说，就要调用 pthread_cond_init 函数来执行初始化。该函数的形式为：

```
int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);
```

参数 attr 是一个条件变量属性对象，如果将 NULL 传递给 attr，则初始化一个具有默认属性的条件变量，否则，就要用与线程属性对象类似的方式，先创建一个条件变量属性对象，再设置它。如果成功，pthread_cond_init 返回 0，如果不成功，pthread_cond_init 就返回一个非零的错误码。下表列出了 pthread_cond_init 的错误码和原因：

错误	原因
EAGAIN	系统缺乏初始化*mutex 所需的非内存资源
ENOMEM	系统缺乏初始化*mutex 所需的内存资源

表 3.9 pthread_cond_init 的错误码和原因

函数 pthread_cond_destroy 销毁一个条件变量。该函数的形式为：

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

如果成功，pthread_cond_destroy 就返回 0。如果不成功，它就返回一个非零的错误码。

3.3.2.2 等待和通知条件变量

条件变量是与断言或条件测试一同调用的，条件变量这个名称就是从这个事实中引申出来的。通常，线程会对一个断言进行测试，如果测试失败，就调用 pthread_cond_wait。函数 pthread_cond_timedwait 可以用来等待一段有限的时间。这两个函数的第一个参数 cond 是一个指向条件变量的指针，第二个参数 mutex 是一个指向互斥量的指针。线程在调用等待条件变量的函数之前，应该拥有这个互斥量。当线程被放置在条件变量的等待队列中时，等待操作会释放这个互斥量。pthread_cond_timedwait 函数的第三个参数是一个指向返回时间的指针，如果条件变量信号没有在此之前出现的话，线程就在这个时间返回。注意，这个值表示的是绝对时间，而不是时间间隔。这两个函数的形式为：

```
int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);
```

如果成功，这两个函数返回 0。如果不成功，它们返回一个非零的错误码。如果 abstime 指定的时间已经到期了，pthread_cond_timedwait 就返回 ETIMEDOUT。

当另一个线程修改了可能会使断言成真的变量时，它应该唤醒一个或多个在等待断言成真的线程。pthread_cond_signal 函数只唤醒一个阻塞在 cond 指向的条件变量上的线程。pthread_cond_broadcast 函数解除所有阻塞在 cond 指向的条件变量上的线程。这两个函数的形式为：

```
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

如果成功，这两个函数返回 0，如果不成功，它们返回一个非零的错误码。等待条件变量的线程在被唤醒后会等待对应的互斥量，在它获得了互斥量之后，它才会被复原成就绪状态。

3.3.2.3 条件变量应用实例

代码 3.5 条件变量应用实例

```
#include <stdio.h>
#include <pthread.h>

int i = 0;
```

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condvar = PTHREAD_COND_INITIALIZER;

void *threadfunc(void *pvoid)
{
    while(1)
    {
        pthread_mutex_lock(&mutex);
        if(i < 200)
        {
            i++;
            pthread_cond_signal(&condvar);
            pthread_mutex_unlock(&mutex);
        }
        else
        {
            pthread_mutex_unlock(&mutex);
            break;
        }
    }

    return NULL;
}

int main()
{
    pthread_t tid;
    pthread_create(&tid, NULL, &threadfunc, NULL);

    pthread_mutex_lock(&mutex);
    while(i < 100)
    {
        pthread_cond_wait(&condvar, &mutex);
    }
    printf("i = %d\n", i);
    pthread_mutex_unlock(&mutex);
    pthread_join(tid, NULL);
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&condvar);
    return 0;
}
```

这是一个简单的条件变量应用程序。主线程监视全局变量 i 的值，如果 i 小于 100，则等待。

子线程递增 i 的值，直到 i 等于 200。主线程直到 i 大于或等于 100 之后，才能继续执行。

运行该程序您会发现，程序的输出结果可能是“i = 100”，也可能是“i = xxx”，其中 xxx 是一个 100 到 200 之间的数。这是因为子线程每次唤醒条件变量并释放互斥量之后，将与主线程一同竞争互斥量。当 i 大于 100 时，如果主线程获得互斥量，就会显示 i 的值。也就是说，等待条件变量的线程在被唤醒时，并不自动获得互斥量，请读者注意。

3.3.3 信号（Signal）处理

信号(signal)是向进程发送的软件通知，通知进程有事件发生。引发信号的事件发生时，信号就被生成(generate)了。进程根据信号采取行动时，信号就被传递(deliver)了。信号的寿命(lifetime)就是信号的生成和传递之间的时间间隔。已经生成但还未被传递的信号被称为挂起(pending)的信号。在信号生成和信号传递之间可能会有相当长的时间。传递信号时，进程必须在处理器上运行。

如果在传递信号时，进程执行了信号处理程序(signal handler)，那么进程就捕捉(catch)到了这个信号。程序可以使用用户编写的函数名作为参数调用 sigaction 来安装用户自定义的信号处理程序；或者以 SIG_DFL 或 SIG_IGN 作为参数调用 sigaction 函数，SIG_DFL 表示采取默认的动作，SIG_IGN 表示忽略信号，这两个动作都不是在“捕捉”信号。如果将进程设置为忽略(ignore)某个信号，那么在传递时那个信号就会被丢弃，不会对进程产生影响。

信号生成时所采取的动作取决于那个信号当前使用的信号处理程序和进程信号掩码(process signal mask)。信号掩码中包含一个当前被阻塞信号(blocked signal)的列表。阻塞一个信号很容易和忽略一个信号混淆起来。被阻塞的信号不会像被忽略的信号一样被丢弃。如果一个挂起信号被阻塞了，那么当进程解除了对那个信号的阻塞时，信号就会被传递出去。程序通过调用 sigprocmask 改变它的进程信号掩码来阻塞一个信号，而通过调用 sigaction 将信号处理程序设置为 SIG_IGN 来忽略一个信号。

注意,信号最初是在多进程系统中引入,而本节的重点在于讲述多线程系统中如何处理信号。本节所涉及的函数、数据类型和宏定义都需要使用 signal.h 头文件。

3.3.3.1 产生信号

每个信号都有一个以 SIG 开头的符号名。信号的名字都定义在 signal.h 中，任何一个使用了信号的 C 程序中都要包含这个文件。信号的名字都是某个大于 0 的小整数的宏定义。表 3.10 中描述了必须的 POSIX 信号,并列出了它们的默认行为。有两个信号 SIGUSR1 和 SIGUSR2 是提供给用户使用的,没有预先指定的用途。出现某些错误时,会产生诸如 SIGFPE 或 SIGSEGV 这样的信号，其它的信号是由 alarm 这样特殊的调用产生的。

信号	描述	默认行为
SIGABRT	进程放弃	与实现有关
SIGALRM	报警时钟	为正常终止
SIGBUS	访问了内存对象中的未定义部分	与实现有关

SIGCHLD	子进程被终止、停止或继续	忽略
SIGCONT	如果进程被停止了,本信号使进程继续执行	继续
SIGFPE	算术计算中出现了被零除的错误	与实现有关
SIGHUP	在控制终端或进程上挂起或终止	非正常终止
SIGILL	无效的硬件指令	与实现有关
SIGINT	交互终端提示信号(通常是Ctrl-C)	非正常终止
SIGKILL	终止(不能被捕获或忽略)	非正常终止
SIGPIPE	向一个没有读程序的管道写入	非正常终止
SIGQUIT	交互终端终止：信息转储(通常为Ctrl-L)	与实现有关
SIGSEGV	无效的内存引用	与实现有关
SIGSTOP	执行停止(不能被捕捉或忽略)	停止
SITTERM	终止	非正常终止
SIGTSTP	终端停止	停止
SIGTTIN	后台进程试图进行读操作	停止
SIGTTOU	后台进程试图进行写操作	停止
SIGURG	在套接字上有高带宽数据	忽略
SIGUSR1	用户定义的信号 1	非正常终止
SIGUSR2	用户定义的信号 2	非正常终止

表 3.10 POSIX 信号

调用 kill 函数可以向指定进程发送指定的信号，调用 raise 函数可以使进程向自己发送指定的信号，调用 alarm 函数可以使进程向自己在经过指定的秒数之后发送 SIGALRM 信号。由于这些函数不属于本节的主题，在此不详细介绍，感兴趣的读者可以自己查询这些函数的相关资料。

3.3.3.2 对信号掩码和信号集进行操作

如前所述，进程可以通过阻塞信号暂时地阻止信号的传递。在传递之前，被阻塞的信号不会影响进程的行为。进程的信号掩码(signal mask)给出了一个信号集合，对哪些信号进行阻塞需要通过信号掩码进行设置。信号掩码的类型为 sigset_t。

信号集由下面的五个函数来操作。每个函数的第一个参数都是一个指向 sigset_t 的指针。sigaddset 负责将 signo 加入信号集，而 sigdelset 将 signo 从信号集中删除。sigemptyset 函数对一个 sigset_t 对象进行初始化，使其不包含任何信号。sigfillset 也可用来对一个 sigset_t 对象进行初始化，使其包含所有的信号。sigismember 负责报告 signo 是否在 sigset_t 中。这五个函数的形式为：

```
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigismember(const sigset_t *set, int signo);
```

对于进程来说，可以通过调用 sigprocmask 函数来检查或修改它的进程信号掩码，由于该函数不属于本节的主题，在此不详细介绍，感兴趣的读者请自己查阅相关资料。

3.3.3.3 信号的捕捉、忽略和等待

sigaction 函数允许调用程序检查或指定与特定信号相关的动作，即处理信号的行为，或忽略信号。由于该函数不属于本节的主题，在此不详细介绍，感兴趣的读者请自己查阅相关资料。

关于等待信号，信号机制提供了一种不需要忙等(busy waiting)的等待事件的机制。忙等是指连续地使用 CPU 来检测事件的发生，而更有效的方式是将进程或线程挂起直到所等待的事件发生为止，这样其它的进程或线程就可以更有效地使用 CPU 了。POSIX 中的 pause，sigsuspend 和 sigwait 函数提供了三种机制，用来挂起进程，直到信号发生为止。

pause 函数将调用线程挂起，直到传递了一个信号为止，这个信号的动作或者是执行用户定义的处理程序，或者是终止进程。如果信号的动作是终止进程，pause 就不返回。如果信号被进程捕捉，pause 就会在信号处理程序返回之后返回。它的形式为：

```
int pause(void);
```

pause 函数总是返回-1。如果被信号中断，pause 就将 errno 设置为 EINTR。

pause 函数面临的一个主要问题是，它将错过在调用之前程序收到的信号。为了解决这个问题，程序必须不间断地执行两项操作——解除对信号的阻塞，并启动 pause。也就是说，这两项操作应该是原子的。sigsuspend 函数即用来解决这个问题。它的形式为：

```
int sigsuspend(const sigset_t *sigmask);
```

sigsuspend 函数用参数 sigmask 指向的信号掩码来阻塞相应的信号，并将进程或线程挂起，直到进程或线程捕捉到相应的信号。sigsuspend 函数在信号处理程序返回之后返回，并将 sigmask 指向的信号掩码设置为调用 sigsuspend 之前的信号阻塞状态。sigsuspend 函数总是返回-1。如果被信号中断，sigsuspend 就将 errno 设置为 EINTR。

sigwait 函数的形式为：

```
int sigwait(const sigset_t *restrict sigmask, int *restrict signo);
```

sigwait 函数会将调用的线程挂起，直到 sigmask 指定的信号集中的信号被挂起，然后从挂起的信号集合中删除那个信号。当 sigwait 返回时，会把从挂起的信号集合中删除的那个信

号的信号码存储在 signo 指向的地址中。如果成功，sigwait 返回 0，如果不成功，sigwait 返回-1 并设置 errno。

要注意的是，sigmask 指定的信号集中的信号必须在 sigwait 调用之前被阻塞，并且不能被忽略的，如果为这些信号指定处理函数，这些处理函数也不会被调用。

3.3.3.4 多线程程序中的信号

进程中的所有线程都共享进程中的信号处理程序，但每个线程可以有它自己的信号掩码。由于线程的操作可以异步于信号，所以线程与信号的交互会比较复杂。对于线程而言，有三种信号的类型，其中异步信号是指传递给某些解除了对该信号的阻塞的线程的信号，同步信号是指传递给引发该信号的线程的信号，定向的信号是指由 pthread_kill 函数发送给指定线程的信号。SIGFPE(浮点异常)这样的错误信号就是同步于引发它们的线程的，因为引发这些信号的线程将等待信号处理程序完成之后才能继续执行，而其它信号因为不与特定的线程相关，所以它们是异步的。如果有几个线程都解除了对同一个异步信号的阻塞，当有信号到达时，线程运行系统就从中挑选一个来处理信号。

pthread_kill 函数将指定的信号发送到指定的线程。它的形式为：

```
int pthread_kill(pthread_t thread, int sig);
```

如果成功，pthread_kill 就返回 0。如果不成功，pthread_kill 就返回一个非零的错误码。下表列出了 pthread_kill 可能发生的错误和对应得错误码。

错误	原因
EINVAL	sig 是无效的或不被支持的信号码
ESRCH	没有线程对应于指定的 ID

表 3. 11 pthread_kill 可能发生的错误和对应得错误码

尽管 pthread_kill 将信号发送到指定的线程，但处理信号的行为可能会影响整个进程。例如，将 SIGKILL 信号发送给指定线程，将使整个进程终止。因为 SIGKILL 信号不能被捕捉，阻塞或忽略，而它的行为就是将进程终止。

虽然信号处理程序是进程范围的，但是每个线程可以有它自己不同的信号掩码。线程可以用 pthread_sigmask 函数来检查或设置它的信号掩码，这个函数是 sigprocmask 在线程化程序中的推广。当进程中有多个线程时，就不应该使用 sigprocmask 函数了，但在创建其它线程之前，也可以被主线程调用，这样可以初始化子线程的信号掩码，因为当子线程被创建时将继承父线程的信号掩码。这个函数的形式为：

```
int pthread_sigmask(int how, const sigset_t *restrict set, sigset_t *restrict oset);
```

参数 how 指定修改信号掩码的方式，how 的值如果为 SIG_SETMASK 会使线程信号掩码被 set 取代，如果为 SIG_BLOCK 会使线程阻塞 set 中的信号，如果为 SIG_UNBLOCK 会使线程解除对

set 中的信号的阻塞。set 为指定的信号集，如果为 NULL，则不进行修改。oset 保存 pthread_sigmask 调用之前的信号掩码。

在多线程的进程中进行信号处理的一种推荐策略是：为信号处理使用特定的线程。主线程在创建线程之前阻塞所有的信号，这样，所有的线程都将信号阻塞了。然后，专门用来处理信号的线程对那些需要处理的信号执行 sigwait，这样指定的信号都将被这个信号处理线程处理。

3.3.3.5 多线程程序中应用信号的实例

下面是一个简单的信号应用实例。类似于前面的多线程实例，主线程在创建子线程之后，等待子线程运行结束，然后继续运行。不同之处在于，主线程没有调用 pthread_join 来等待子线程运行结束，而是等待子线程发送的信号。子线程则在运行结束前向主线程发送信号，来通知主线程。这样，利用信号机制，实现了线程的同步。

代码 3. 6 多线程程序中应用信号的实例

```
#include <stdio.h>
#include <pthread.h>
#include <signal.h>

void *threadfunc(void *pvoid)
{
    pthread_t* main_tid = (pthread_t*)pvoid;
    printf("Child thread says:Hello!World!\n");
    pthread_kill(*main_tid, SIGUSR1);
}

int main()
{
    sigset_t sigs;
    int sig;
    pthread_t tid;
    pthread_t main_tid;

    sigemptyset(&sigs);
    sigaddset(&sigs, SIGUSR1);
    sigprocmask(SIG_BLOCK, &sigs, NULL);

    main_tid = pthread_self();
    pthread_create(&tid, NULL, &threadfunc, &main_tid);

    sigwait(&sigs, &sig);
```

```
    printf("Main thread says:Hello!World!\n");

    return 0;
}
```

3.3.4 读写锁（read-write lock）

在一些程序中存在一个称作读者写者的问题,即对于某些资源的访问,存在两种可能的情况，一种是访问必须是排他的，称作写操作，另一种是访问可以是共享的，称作读操作。显而易见，这一问题和相应的表述是从对文件的读写操作中引申出来的。

处理读者写者问题的两种常见的策略是：强读者同步(strong reader synchronization)和强写者同步(strong writer synchronization)。在强读者同步中，总是给读者以优先权，只要写者当前没有进行写操作，读者就可以获得访问权。在强写者同步中，通常将优先权交给写者,而将读者延迟到所有等待的或活动的写者都完成为止。由于读者都需要最新的信息，所以航线预订系统会使用强写者同步，而图书馆的参考数据库可能会采用强读者同步。

POSIX 提供了读写锁的机制，如果写者没有持有锁，就允许所有读者获取这个锁，而当有写者阻塞在锁上时，就由实现 POSIX 的系统来决定是否允许读者获取锁。

3.3.4.1 初始化和销毁读写锁

读写锁由 pthread_rwlock_t 类型的变量表示。程序在使用 pthread_rwlock_t 变量进行同步之前，必须调用 pthread_rwlock_init 函数来初始化这个变量。这个函数的形式为：

```
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);
```

参数 rwlock 是一个指向读写锁的指针，参数 attr 是一个读写锁属性对象的指针，如果将 NULL 传递给它，则使用默认属性来初始化一个读写锁。如果成功，pthread_rwlock_init 就返回 0。如果不成功，pthread_rwlock_init 就返回一个非零的错误码。下表列出了，pthread_rwlock_init 可能发生的错误和对应得错误码。

错误	原因
EAGAIN	系统缺乏初始化*rwlock 所需的非内存资源
ENOMEM	系统缺乏初始化*rwlock 所需的内存资源
EPERM	调用程序没有适当的特权

表 3.12 pthread_rwlock_init 可能发生的错误和对应得错误码

pthread_rwlock_destroy 函数用于销毁一个读写锁。它的形式为：

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

参数 rwlock 为指向要销毁的读写锁的指针。可以用 pthread_rwlock_init 对已经被销毁的

读写锁重新进行初始化。如果成功，pthread_rwlock_destroy 就返回 0。如果不成功，pthread_rwlock_destroy 就返回一个非零的错误码。

3.3.4.1 操作读写锁

有五个函数用来操作读写锁。

pthread_rwlock_rdlock 和 pthread_rwlock_tryrdlock 函数用来为读操作获取一个读写锁。pthread_rwlock_wrlock 和 pthread_rwlock_trywrlock 函数用来为写操作获取一个读写锁。pthread_rwlock_rdlock 和 pthread_rwlock_wrlock 阻塞线程，直到获取读写锁，而 pthread_rwlock_tryrdlock 和 pthread_rwlock_trywrlock 则会立即返回。pthread_rwlock_unlock 函数会将锁释放掉。这些函数的形式为：

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

如果成功，这些函数返回 0，如果不成功，这些函数就返回一个非零的错误码。如果因为锁已经被持有而无法获取，pthread_rwlock_tryrdlock 和 pthread_rwlock_trywrlock 就返回 EBUSY。

3.3.4.1 读写锁应用实例

下面是一个简单的读写锁应用实例。程序中有一个读写锁的全局变量，供各个线程操作。主线程首先获得写锁，然后随机创建一百个读线程或写线程，之后释放锁。此后读线程和写线程开始竞争读写锁。读线程会争取读锁，写线程会争取写锁。读者可以自己运行一下这个程序，然后分析线程的创建顺序和执行顺序的差别，来了解读写锁是怎样协调线程的运行的。

代码 3.7 读写锁应用实例

```
#include <stdio.h>
#define __USE_UNIX98
#include <pthread.h>
#include <stdlib.h>

#define THREADCOUNT 100

pthread_rwlock_t rwlock;

void *reader(void *pvoid)
{
    pthread_rwlock_rdlock(&rwlock);
    printf("reader%d worked.\n", (int)pvoid);
```

```
    if(pthread_rwlock_unlock(&rwlock))
    {
    printf("reader%d unlock error!\n", (int)pvoid);
    }

    return NULL;
}

void *writer(void *pvoid)
{
    pthread_rwlock_wrlock(&rwlock);
    printf("writer%d worked.\n", (int)pvoid);
    if(pthread_rwlock_unlock(&rwlock))
    {
    printf("writer%d unlock error!\n", (int)pvoid);
    }

    return NULL;
}

int main(void)
{
    pthread_t reader_id, writer_id;
    pthread_attr_t threadattr;
    int i, rand;
    int readercount = 1, writercount = 1;
    int halfmax = RAND_MAX / 2;

    if(pthread_rwlock_init(&rwlock, NULL))
    {
    printf("initialize rwlock error!\n");
    }

    pthread_attr_init(&threadattr);
    pthread_attr_setdetachstate(&threadattr, PTHREAD_CREATE_DETACHED);

    pthread_rwlock_wrlock(&rwlock);
    for(i = 0; i < THREADCOUNT; i++)
    {
    rand = random();
    if(rand < halfmax)
    {
        pthread_create(&reader_id, &threadattr, reader, (void*)readercount);
        printf("Created reader%d\n", readercount++);
```

```
    }
    else
    {
        pthread_create(&writer_id, &threadattr, writer, (void*)writercount);
        printf("Created writer%d\n", writercount++);
    }
    }
    pthread_rwlock_unlock(&rwlock);

    pthread_exit(NULL);
    return 0;
}
```

6.3.5 信号量（Semaphore）

1965 年，E.W.Dijkstra 提出了信号量的概念，之后信号量即成为操作系统实现互斥和同步的一种普遍机制。信号量是包含一个非负整型变量，并且带有两个原子操作 wait 和 signal。wait 还可以被称为 down、P 或 lock，signal 还可以被称为 up、V、unlock 或 post。

如果信号量的非负整形变量 S 大于零，wait 就将其减 1，如果 S 等于 0，wait 就将调用线程挂起。对于 signal 操作，如果有线程在信号量上阻塞(此时 S 等于 0)，signal 就会解除对某个等待线程的阻塞，使其从 wait 中返回，如果没有线程阻塞在信号量上，signal 就将 S 加 1。

由此可见，S 可以被理解为一种资源的数量，信号量即是通过控制这种资源的分配来实现互斥和同步的。如果把 S 设为 1，信号量即可实现互斥量的功能。如果 S 的值大于 1，那么信号量即可使多个线程并发运行。另外，信号量不仅允许使用者申请和释放资源，而且还允许使用者创造资源，这就赋予了信号量实现同步的功能。可见，信号量的功能要比互斥量丰富许多。

POSIX 信号量是一个 sem_t 类型的变量，但 POSIX 有两种信号量的实现机制：无名信号量和命名信号量。无名信号量可以用在共享内存的情况下，比如实现进程中各个线程之间的互斥和同步。命名信号量通常用于不共享内存的情况下，比如不共享内存的进程之间。

使用本节所介绍的函数和数据结构需要引用头文件 semaphore.h。

6.3.5.1 POSIX 无名信号量

在使用信号量之前，必须对其进行初始化。sem_init 函数初始化指定的信号量，它的形式为：

```
int sem_init(sem_t *sem, int pshared, unsigned value);
```

参数 sem 指向要初始化的信号量，参数 value 为信号量的初始值。参数 pshared 用于说明信号量的共享范围，如果 pshared 为 0，那么该信号量只能由初始化这个信号量的进程中的线程使用，如果 pshared 非零，任何可以访问到这个信号量的进程都可以使用这个信号量。如果成功,sem_init 返回 0,如果不成功,sem_init 返回-1 并设置 errno。下表列出了,sem_init 可能发生的错误和对应得错误码。

错误	原因
EINVAL	value 大于 SEM_VALUE_MAX
ENOSPC	初始化资源已经耗尽，或者信号量的数目超出了 SEM_NSEMS_MAX 的范围
EPERM	调用程序没有适当的特权

表 3.13 sem_init 可能发生的错误和对应得错误码

函数 sem_destroy 销毁一个指定的信号量，它的形式为：

```
int sem_destroy(sem_t *sem);
```

参数 sem 为指向要销毁的信号量的指针。如果成功，sem_destroy 返回 0，如果不成功，sem_destroy 返回-1 并设置 errno。如果*sem 不是有效的信号量，sem_destroy 就将 errno 置为 EINVAL。

sem_post 函数实现对指定信号量的 signal 操作，它的形式为：

```
int sem_post(sem_t *sem);
```

如果成功，sem_post 返回 0。如果不成功，sem_post 返回-1 并设置 errno。如果*sem 不是有效的信号量，sem_post 就将 errno 置为 EINVAL。

sem_wait 函数实现对指定信号量的 wait 操作。sem_trywait 函数与 sem_wait 类似，只是在试图对一个为零的信号量进行操作时，它不会阻塞调用线程，而是立即返回，类似于互斥量操作中的 pthread_mutex_trylock()。这两个函数的形式为：

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

如果成功，这两个函数返回 0，如果不成功，这些函数返回-1 并设置 errno。如果*sem 不是有效的信号量，sem_ wait 就将 errno 置为 EINVAL。如果 sem_trywait 在信号量为零时执行 wait 操作，则将 errno 置为 EAGAIN。

sem_post，sem_wait 和 sem_trywait 同样可用于命名信号量。

3.3.5.2 POSIX 命名信号量

之所以称为命名信号量，是因为它有一个名字、一个用户 ID、一个组 ID 和权限，这些是提供给不共享内存的那些进程使用命名信号量的接口。命名信号量的名字是一个遵守路径名构造规则的字符串。

sem_open 函数用于创建或打开一个命名信号量。它的形式为：

```
sem_t *sem_open(const char *name, int oflag);
```

参数 name 是一个标识信号量的字符串。参数 oflag 用来确定是创建信号量还是连接已有信号量。如果设置了 oflag 的 O_CREAT 比特位，则会创建一个新的信号量。

sem_close 函数用于关闭命名信号量。它的形式为：

```
int sem_close(sem_t *sem);
```

参数 sem 是指向要关闭的信号量的指针。单个程序可以用 sem_close 函数关闭命名信号量，但是这样做并不能将信号量从系统中删除，因为命名信号量在单个程序的执行之外是具有持久性的。当进程调用_exit、exit、exec 或从 main 返回时，进程打开的命名信号量同样会被关闭。如果成功，sem_ close 返回 0，如果不成功，sem_ close 返回-1 并设置 errno。如果*sem 不是有效的信号量，sem_ close 就将 errno 置为 EINVAL。

sem_unlink 函数用于在所有进程关闭了命名信号量之后，将信号量从系统中删除。它的形式为：

```
int sem_unlink(const char *name);
```

参数 name 为要删除的命名信号量的名字。如果成功，sem_ unlink 返回 0，如果不成功，sem_ unlink 返回-1 并设置 errno。下表列出了，sem_ unlink 可能发生的错误和对应得错误码。

错误	原因
EACCES	权限不正确
ENAMETOOLONG	name 比 PATH_NAME 长，或者它有一个组件超出了 NAME_MAX 的范围
ENOENT	信号量不存在

表 3.14 sem_ unlink 可能发生的错误和对应得错误码

3.3.5.3 信号量应用实例

下面是一个无名信号量的应用实例，它完成的工作为：创建两个线程，这两个线程各自将自己的一个整型变量 i 从 1 递增到 100，并通过信号量控制递增的过程，即这两个整型变量的差不能超过 5。

代码 3.7 信号量应用实例

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define UPBOUND 100
sem_t sem1;
sem_t sem2;

void *threadfunc1(void *pvoid)
{
    int i = 0;
    while(i < UPBOUND)
    {
        sem_wait(&sem1);
        i++;
        printf("The integer of child thread1 is %d now.\n", i);
        sem_post(&sem2);
    }
}

void *threadfunc2(void *pvoid)
{
    int i = 0;
    while(i < UPBOUND)
    {
    sem_wait(&sem2);
    i++;
    printf("The integer of child thread2 is %d now.\n", i);
    sem_post(&sem1);
}
return (NULL);
}
int main()
{
    pthread_t tid1, tid2;

    int i = sem_init(&sem1, 0, 5);
    printf("%d\n", i);
    sem_init(&sem2, 0, 5);
    printf("Semaphores are initialized.\n");

    pthread_create(&tid1, NULL, &threadfunc1, NULL);
    pthread_create(&tid2, NULL, &threadfunc2, NULL);
```

```
    printf("Threads are started.\n");

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("Threads finished.\n");

    sem_destroy(&sem1);
    sem_destroy(&sem2);
    printf("Semaphores are destoried.\n");

    printf("Main thread finished\n");
    return 0;
}
```

第四章 Windows 多线程编程

[引言]本章主要介绍在 Windows 操作系统下进行多线程编程的基本接口及编程技术。

4 .1 Windows 操作系统的一些基本知识：

Windows 是 Microsoft 公司在 1983 年 11 月宣布，并在两年之后的 1985 年 11 月推出的图形操作系统。1993 年 7 月，Microsoft 公司推出了支持 32 位抢占式多任务、多线程的 Windows NT，并于 1995 年 8 月推出了 Windows 95，使 Windows 一举成为最流行的图形操作系统，尤其在 PC 机上得到了广泛的应用。

对于程序员来说，操作系统是由本身的 API（Application Programming Interface）定义的。API 包含了所有应用程序构造操作系统的函数调用，同时包含了相关的数据类型和结构。在 Windows 中，API 还意味着一个特殊的程序体系结构。Windows 从 1.0 到 3.1 使用的是英特尔 8086、8088 和 286 微处理器的 16 位指令，而从 Windows NT 和 Windows 95 开始，Windows 开始支持使用英特尔 386、486 和 Pentium 处理器的 32 位指令。用于 16 位版本 Windows 的 API 现在被称作 Win16，用于 32 位版本 Windows 的 API 现在被称作 Win32。由于当前普遍使用的是 32 位版本的 Windows，因此本书只介绍 Win32 的相关内容。

在介绍 Windows API 之前，首先要讲述一下内核对象以及句柄的概念。内核对象是由操作系统内核分配的，只能由内核访问的一个内存块，用来供系统和应用程序使用和管理各种系统资源。作为一个 Windows 软件开发人员，你经常需要创建、打开和操作各种内核对象，包括符号对象、事件对象、文件对象、文件映射对象、I/O 完成端口对象、作业对象、信箱对象、互斥量、管道对象、进程对象、信标对象、线程对象和等待计时器对象等。这些对象都是通过调用函数来创建的。例如，CreateThread 函数可使系统能够创建一个线程对象。不同的内核对象拥有不同的数据结构，它的成员负责维护该对象的各种信息。由于内核对象的数据结构只能被内核程序访问，因此应用程序无法在内存中找到这些数据结构并直接改变它们的内容。Windows 规定了这个限制条件，目的是为了确保内核对象结构保持状态的一致。这个限制也使 Windows 能够在不破坏任何应用程序的情况下在这些结构中添加、删除和修改数据成员。

如果我们不能直接改变这些数据结构，那么我们的应用程序如何才能操作这些内核对象呢？解决办法是，Windows 提供了一组函数，以便用定义的很好的方法来对这些结构进行操作。这些内核对象始终都可以通过这些函数进行访问。当调用一个用于创建内核对象的函数时，该函数就返回一个用于标识该对象的句柄。该句柄可以被视为一个不透明的值，你的进程中

的任何线程都可以使用这个值，将这个句柄传递给 Windows 的各个函数，这样，系统就能知道你想操作哪个内核对象了。每个进程被初始化时，系统为它分配一个句柄表，用于保存该进程使用的内核对象的信息，而句柄值则是相应内核对象在句柄表中的索引值，因此句柄值是进程相关的，相同的句柄值在不同的进程中可能标识不同的内核对象，也就是说，句柄是相对于进程的，这使得对内核对象的使用更加安全，操作系统更加健壮。

内核对象由内核拥有，而不是由进程拥有，因此进程是可以共享内核对象的，一个进程中止执行，它使用的内核对象并不一定会被撤销。内核知道有多少进程正在使用某个内核对象，因为每个对象都包含一个使用计数，每当有一个进程在使用该内核对象时，其使用计数就加 1，而使用该内核对象的进程中止时，其使用计数就减 1，当使用计数为 0 时，操作系统才撤销该内核对象。

4.2 Win32 API 的线程库：

4.2.1 创建线程的基本问题：

线程可以由进程中的任意线程创建，而进程的主线程在进程装载时自动创建。每个线程都有自己的进入点函数。主线程的进入点函数的名称必须为 main、wmain、WinMain、wWinMain 之一。这是编写 Windows 应用程序的规范，其中各个进入点函数对应于不同类型的 Windows 应用程序，其对应关系见表 4.1：

进入点	应用程序类型
WinMain	需要 ANSI 字符和字符串的 GUI 应用程序
wWinMain	需要 Unicode 字符和字符串的 GUI 应用程序
Main	需要 ANSI 字符和字符串的 CUI 应用程序
Wmain	需要 Unicode 字符和字符串的 CUI 应用程序

表 4.1 编写 Windows 应用程序的规范

当你想要在进程中创建一个线程时，也必须给这个线程提供一个进入点函数，它的格式应该有类似下面函数的格式：

```
DWORD WINAPI ThreadFunc(PVOID pvParam){
    DWORD dwResult = 0;
    ….
    ….
    ….
    return(dwResult);
}
```

下面对线程函数的几个问题作一下说明：

•主线程的进入点函数的名字必须是 main、wmain、WinMain 或 wWinMain，而其它线程函数可以使用其它的任何名字。

•线程函数必须返回一个值，它将成为该线程的退出代码。这与 C/C++运行期库（run-time library）关于让主线程的退出代码作为线程的退出代码的原则是相似的。

•线程函数应该尽可能使用函数参数和局部变量。当使用静态变量和全局变量时，多个线程可以同时访问这些变量，这可能破坏变量的内容，而参数和局部变量是在线程堆栈中创建的，因此它们不太可能被另一个线程破坏。

4.2.2 创建线程的 API：

Windows 为创建线程提供了名为 CreateThread 的 API 函数，如果想要创建一个或多个线程，只要在一个已经在运行的线程调用这个函数即可。CreateThread 函数的格式如下：

```
HANDLE CreateThread(
    PSECURITY_ATTRIBUTES psa,
    DWORD cbStack,
    PTHREAD_START_ROUTINE pfnStartAddr,
    PVOID pvParam,
    DWORD fdwCreate,
    PDWORD pdwThreadID);
```

当 CreateThread 被调用时，系统创建一个线程内核对象。该线程内核对象不是线程本身，而是操作系统用来管理线程的较小的数据结构。可以将线程内核对象视为由关于线程的统计信息组成的一个小型数据结构。这与进程和进程内核对象之间的关系是相同的。之后系统从进程的地址空间中分配内存，供线程的堆栈使用。新线程运行的进程环境与创建线程的环境相同。因此，新线程可以访问进程的内核对象的所有句柄、进程中的所有内存和在这个相同的进程中的所有其它线程的堆栈。这使得单个进程中的多个线程能够非常容易地互相通信。

下面介绍一下 CreateThread 函数的各个参数：

1、psa
psa 参数是指向 SECURITY_ATTRIBUTES 结构的指针。如果想要该线程内核对象的默认安全属性，可以（并且在通常情况下）传递 NULL。如果希望所有的子进程能够继承该线程对象的句柄，必须设定一个 SECURITY_ATTRIBUTES 结构，它的 bInheritHandle 成员被初始化为 TRUE。

2、cbStack
cbStack 参数用于设定线程可以将多少地址空间用于它自己的堆栈。如果将 0 作为这个参数传递给 CreateThread，线程的堆栈大小将由链接程序中的/STACK 开关的值来确定。/STACK 开关的形式为：/STACK：[reserve][，commit]。其中 reserve 参数用于设定系统默认的线程堆栈最大值，默认为 1MB；commit 参数用于设定线程启动时首先为线程分配的内存量，以

页为单位，默认为 1 页。而当 cbStack 的值大于 0 时，链接程序将比较 cbStack 和 reserve 的值，谁大就用谁。线程堆栈是在线程运行过程中以页为单位动态分配的，这使得线程堆栈能够根据需要动态地扩大，而这个扩大的过程是有上限的，当它超过设定的线程堆栈最大值时，将会出现一个堆栈溢出错误，这防止了应用程序用完大量的内存，也可以使程序员检测出程序中不合理的内存分配。

3、pfnStartAddr 和 pvParam

pfnStartAddr 参数用于指明想要新线程执行的线程函数的地址。线程函数的 pvParam 参数与传递给 CreateThread 的 pvParam 参数是相同的，CreateThread 函数正是使用该参数为新线程传递参数的。该参数及可以是一个数字值，也可以是指向包含其它信息的一个数据结构的指针。

4、fdwCreate

fdwCreate 参数可以设定用于控制创建线程的其他标志。它可以是两个值中的一个。如果该值是 0，那么线程创建后可以立即进行调度。如果该值是 CREATE_SUSPENDED，系统可以完整地创建线程并对它进行初始化，但是要暂停该线程的运行，这样它就无法参加调度了，这使得应用程序能够在新线程运行之前修改线程的某些属性。

5. pdwThreadID

pdwThreadID 是 CreateThread 的最后一个参数，它必须是 DWORD 的一个有效地址，CreateThread 使用这个地址来存放系统分配给新线程的 ID。线程 ID 是系统赋予线程的在系统范围内的唯一的标识符。在这里应该强调的是，使用 ID 来跟踪线程是存在风险的，因为虽然系统不会为其它线程赋予与被跟踪线程相同的 ID，但在被跟踪线程终止后，系统就可以将这个 ID 赋予其它线程了，这将导致预料之外的错误。因此在通常情况下，我们以 NULL 作为 pdwThreadID 的值，这样就告诉 CreateThread，你对线程的 ID 不感兴趣。

在 C/C++多线程运行期库中还有另一个创建线程的函数_beginthreadex:

```
unsigned long _beginthreadex(
        void *security,
        unsigned stack_size,
        unsigned (*start_address)(void *),
        void *arglist,
        unsigned initflag,
        unsigned *thrdaddr);
```

细心的读者可以发现，_beginthreadex 与 CreateThread 的参数实际上是相同的，那么他们有什么内在的区别呢？这里首先要讲解一下 C/C++运行期库的相关内容。

C/C++运行期库为 C/C++应用程序的运行提供一些底层的支持，比如保持一些程序运行状态的信息，决定内存分配的具体方式等。这些支持是通过一些数据结构和函数实现的。在 Visual C++中，配有 6 个 C/C++运行期库，见表 4.2：

库名	描述

LibC.lib	用于单线程应用程序的静态链接库（创建新应用程序时的默认库）
LibCD.lib	用于单线程应用程序的静态链接库的调试版
LibCMt.lib	用于多线程应用程序的静态链接库的发行版
LibCMtD.lib	用于多线程应用程序的静态链接库的调试版
MSVCRt.lib	用于动态链接 MSVCRt.dll 库的发行版的输入库
MSVCRtD.lib	用于动态链接 MSVCRtD.dll 库的调试版的输入库（该库同时支持单线程应用程序和多线程应用程序）

表 4.2Visual C++中的 C/C++运行期库

这些运行期库是随着计算机技术的发展逐步被开发出来的。1970 年标准 C 运行期库问世，但现在它已经不适合某些应用程序的开发了。为了适应这些新技术的要求，新的运行期被逐渐开发出来。具体到多线程问题上，之所以要开发多线程应用程序运行期库并在程序链接时将其链接入多线程应用程序，就是因为随着多线程技术的采用，原来单线程应用程序的运行期库已经不能适应多线程的需要了。早期的单线程运行期库为程序的主线程维护一个数据块，即 tiddata 结构变量，来记录程序运行的一些信息，而多线程程序就需要为每个线程维护一个 tiddata 变量，这就是多线程运行期库对于多线程程序的支持的一个方面。一个很容易理解的例子是：线程需要一个变量在线程发生异常时来保存异常的类型，即 tiddata 变量中的_terrno 变量，如果用单线程运行期库，那么进程中将只存在一个 tiddata 变量，进程中的各个线程共享这个变量,这显然是不合适的,因为不同的线程可能同时发生不同的异常，这将可能导致异常处理程序对异常类型的判断错误。_beginthreadex 函数在创建线程之前会为线程分配数据块，并对这个数据块进行初始化，然后将数据块与线程联系起来，再为线程函数建立结构化异常处理桢来处理线程函数中发生的异常，但 CreateThread 函数并不进行这些工作，这将导致两个问题：1、如果在线程函数中使用到了 tiddata 变量，那么 C/C++运行期库函数可以现场为线程分配一个 tiddata 变量，但当线程退出时，这个变量并不会被清除，从而造成内存泄漏；2、如果线程使用 C/C++运行期库函数 signal 函数，那么整个进程将会终止运行（signal 函数的功能在此不详细描述）。因此，推荐读者使用_beginthreadex 创建线程，因为这能避免一些可能发生的问题，虽然这些问题并不常见。

4.2.3 操作线程的 API

暂停线程:

```
DWORD SuspendThread(HANDLE hThread);
```

任何线程都可以调用该函数来暂停另一个线程的运行，只要它知道对方的句柄。当然，线程也可以自行暂停运行，但是不能自行恢复运行。SuspendThread 返回的是线程的前一个暂停计数。线程暂停是线程内核对象的一个内部值，用于指明线程的暂停计数，最多次数可以是 MAXIMUM_SUSPEND_COUNT 次（在 WinNT.h 中定义为 127）。

在实际环境中，调用 SuspendThread 时必须小心，因为不知道暂停线程运行时它在进行什么操作。如果线程正在试图从堆中分配内存，那么该线程将在该堆上设置一个锁。当其他线程试图访问该堆时，这些线程的访问就被停止，直到第一个线程恢复运行。只有确切知道目标

线程是什么（或者目标线程正在做什么），并且采取强有力的措施来避免因暂停线程的运行而带来的问题或死锁状态，SuspendThread 才是安全的。

恢复线程：

```
DWORD ResumeThread(HANDLE hThread);
```

该函数用于将处于暂停状态的线程置于就绪状态，使其参加线程调度。如果 ResumeThread 运行成功，它将返回线程的前一个暂停计数，否则返回 0xFFFFFFFF。单个线程可以暂停多次，如果一个线程被暂停了 3 次，它必须被恢复 3 次，才能恢复到就绪状态。

使线程睡眠：

```
VOID Sleep(DWORD dwMIlliseconds);
```

该函数可是线程暂停自己的运行，直到 dwMIlliseconds 过去为止。当线程调用 Sleep 函数时，它将自动放弃它剩余的时间片，迫使系统进行线程调度。而线程睡眠的时间也并不一定是 dwMIlliseconds 指定的时间，比如 dwMIlliseconds 中指定线程睡眠 100ms，那么它可能会睡眠 100ms，也可能睡眠几秒甚至几分钟，因为 Windows 不是一个实时操作系统，虽然线程可能在规定的时间被唤醒，但是它能否做到，取决于系统中还有什么操作正在进行。

终止线程：

若要终止线程的运行，可以使用下面的方法：

1、线程函数返回（最好使用这种方法）。

2、通过调用 ExitThread 函数，线程将自行撤消（最好不要使用这种方法）。

3、同一个进程或另一个进程中的线程调用 TerminateThread 函数（应该避免使用这种方法）。

4、包含线程的进程终止运行。

下面将介绍以上各种方法，并说明线程终止运行时会出现什么情况。

1、线程函数返回
始终都应该将线程设计成这样的形式，即当想要线程终止运行时，它们就能够返回。这是确保所有线程资源被正确地清除的唯一办法。

如果线程能够返回，就可以确保下列事项的实现：
- 在线程函数中创建的所有 C + +对象均将通过它们的撤消函数正确地撤消。
- 操作系统将正确地释放线程堆栈使用的内存。
- 系统将线程的退出代码（在线程的内核对象中维护）设置为线程函数的返回值。

• 系统将递减线程内核对象的使用计数。

2、ExitThread 函数

可以让线程调用 ExitThread 函数，以便强制线程终止运行：

VOID ExitThread（DWORD dwExitCode）

该函数将终止线程的运行，并导致操作系统清除该线程使用的所有操作系统资源。但是，C++资源（如 C++类对象）将不被撤消。由于这个原因，最好从线程函数返回，而不是通过调用 ExitThread 来返回。

当然，可以使用 ExitThread 的 dwExitThread 参数告诉系统将线程的退出代码设置为什么。ExitThread 函数并不返回任何值，因为线程已经终止运行，不能执行更多的代码。注意终止线程运行的最佳方法是让它的线程函数返回。但是，如果使用本节介绍的方法，应该知道 ExitThread 函数是 Windows 用来撤消线程的函数。如果编写 C/C++代码，那么决不应该调用 ExitThread，而应该使用 Visual C++运行期库函数_endthreadex，因为_endthreadex 函数将对线程的 tiddata 结构变量占用的内存进行释放，而 ExitThread 函数不进行这项工作，从而可能导致内存泄漏。

3、TerminateThread 函数

调用 TerminateThread 函数也能够终止线程的运行：

```
BOOL TerminateThread(){
    HANDLE hThread;
    DWORD dwExitCode;
}
```

与 ExitThread 不同，ExitThread 总是撤消调用的线程，而 TerminateThread 能够撤消任何线程。hThread 参数用于标识被终止运行的线程的句柄。当线程终止运行时，它的退出代码成为 dwExitCode 参数传递的值。同时，线程的内核对象的使用计数也被递减。注意 TerminateThread 函数是异步运行的函数，也就是说，它告诉系统你想要线程终止运行，但是，当函数返回时，不能保证线程被撤消。如果需要确切地知道该线程已经终止运行，必须调用 WaitForSingleObject(后面线程通信的部分将介绍这个函数)或者类似的函数。

设计良好的应用程序从来不使用这个函数，因为被终止运行的线程收不到它被撤消的通知，线程不能正确地清除，并且不能防止自己被撤消。注意当使用返回或调用 ExitThread 的方法撤消线程时，该线程的内存堆栈也被撤消。但是，如果使用 TerminateThread，那么在拥有线程的进程终止运行之前，系统不撤消该线程的堆栈。Microsoft 故意用这种方法来实现 TerminateThread。如果其它仍然正在执行的线程要引用被强制撤消的线程堆栈上的值，那么其他的线程就会出现访问违规的问题。如果将已经撤消的线程的堆栈留在内存中，那么其他线程就可以继续很好地运行。

4、在进程终止运行时撤消线程

ExitProcess 和 TerminateProcess 函数可以用来终止线程的运行，差别在于这些线程将会使终止运行的进程中的所有线程全部终止运行。另外，由于整个进程已经被关闭，进程使用的所有资源肯定已被清除。这当然包括所有线程的堆栈。这两个函数会导致进程中的剩余线程被强制撤消，就像从每个剩余的线程调用 TerminateThread 一样。显然，这意味着正确的应用程序清除没有发生，即 C++对象撤消函数没有被调用，数据没有转至磁盘等等。

下面是一个线程的内核对象的示意图：

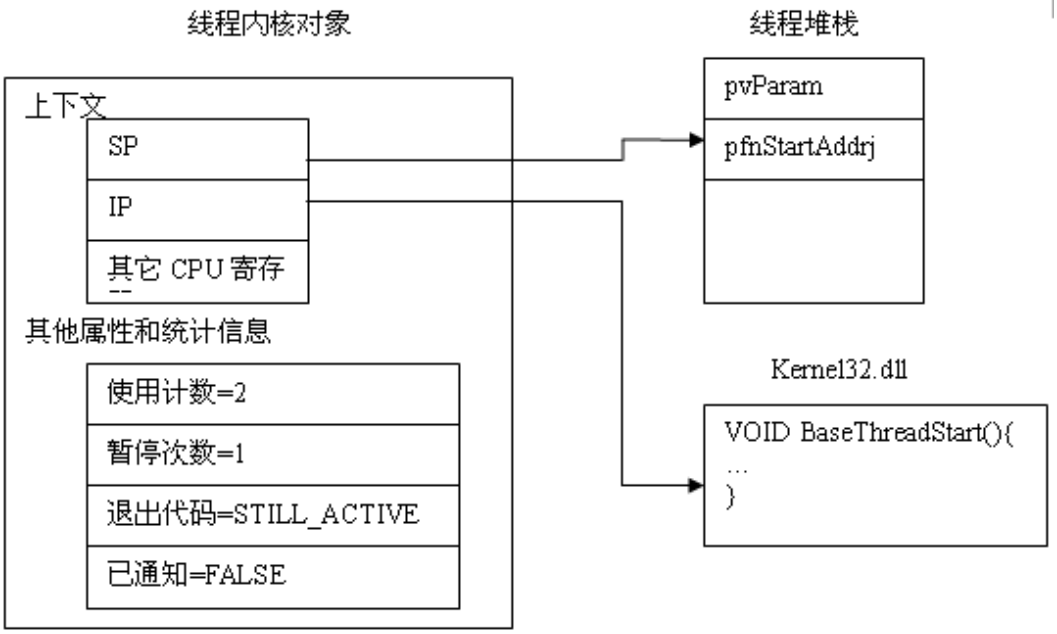

图 4.1 一个线程的内核对象的示意图

图 4.1 显示了系统在创建线程和对线程进行初始化时必须做些什么工作。让我们仔细看一看这个图，以便确切地了解发生的具体情况。调用 C r e a t e T h r e a d 可使系统创建一个线程内核对象。该对象的初始使用计数是 2（在线程停止运行和从 C r e a t e T h r e a d 返回的句柄关闭之前，线程内核对象不会被撤消）。线程的内核对象的其他属性也被初始化，暂停计数被设置为 1，退出代码始终为 STILL_ACTIVE（0x103），该对象被设置为未通知状态。

一旦内核对象创建完成，系统就分配用于线程的堆栈的内存。该内存是从进程的地址空间分配而来的，因为线程并不拥有它自己的地址空间。然后系统将两个值写入新线程的堆栈的上端（线程堆栈总是从内存的高地址向低地址建立）。写入堆栈的第一个值是传递给 CreateThread 的 pvParam 参数的值。紧靠它的下面是传递给 CreateThread 的 pfnStartAddr 参数的值。

每个线程都有它自己的一组 C P U 寄存器，称为线程的上下文。该上下文反映了线程上次运行时该线程的 CPU 寄存器的状态。线程的这组 CPU 寄存器保存在一个 CONTEXT 结构（在 WinNT.h 头文件中作了定义）中。CONTEXT 结构本身则包含在线程的内核对象中。

指令指针和堆栈指针寄存器是线程上下文中两个最重要的寄存器。线程总是在进程的上下文

中运行的。因此，这些地址都用于标识拥有线程的进程地址空间中的内存。当线程的内核对象被初始化时，CONTEXT 结构的堆栈指针寄存器被设置为线程堆栈上用来放置 pfnStartAddr 的地址。指令指针寄存器置为称为 BaseThreadStart 的未文档化（和未输出）的函数的地址中。该函数包含在 Kernel32.dll 模块中（这也是实现 CreateThread 函数的地方）。上图显示了它的全部情况。

下面是 BaseThreadStart 函数执行的基本操作：

```
VOID BaseThreadStart(PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam){
    __try{
        ExitThread((pfnStartAddr)(pvParam));
    }
    __except(UnhandledExceptionFilter(GetExceptionInformation())){
        ExitProcess(GetExceptionCode());
    }
    // NOTE: We never get here.
}
```

当线程完全初始化后，系统就要查看 CREATE_SUSPENDED 标志是否已经传递给 CreateThread。如果该标志没有传递，系统便将线程的暂停计数递减为 0，该线程可以调度到一个进程中。然后系统用上次保存在线程上下文中的值加载到实际的 CPU 寄存器中。这时线程就可以执行代码，并对它的进程的地址空间中的数据进行操作。由于新线程的指令指针被置为 BaseThreadStart，因此该函数实际上是线程开始执行的地方。BaseThreadStart 的原型会使你认为该函数接收了两个参数，但是这表示该函数是由另一个函数来调用的。新线程只是在此处产生并且开始执行。BaseThreadStart 认为它是由另一个函数调用的，因为它可以访问两个参数。但是，之所以可以访问这些参数，是因为操作系统将值显式写入了线程的堆栈（这就是参数通常传递给函数的方法）。注意，有些 CPU 结构使用 CPU 寄存器而不是堆栈来传递参数。对于这些结构来说，系统将在允许线程执行 BaseThreadStart 函数之前对相应的寄存器正确地进行初始化。

当新线程执行 BaseThreadStart 函数时，将会出现下列情况：

• 在线程函数中建立一个结构化异常处理（SEH）帧，这样，在线程执行时产生的任何异常情况都会得到系统的某种默认处理（关于结构化异常处理在此并不详述，大致来讲，它是一种处理程序退出和程序执行异常的机制，由操作系统和编译程序共同支持。SEH 桢即是一段处理代码）。

• 系统调用线程函数，并将你传递给 CreateThread 函数的 pvParam 参数传递给它。

• 当线程函数返回时，BaseThreadStart 调用 ExitThread，并将线程函数的返回值传递给它。该线程内核对象的使用计数被递减，线程停止执行。

• 如果线程产生一个没有处理的异常条件，由 BaseThreadStart 函数建立的 SEH 帧将负责处理该异常条件。通常情况下，这意味着向用户显示一个消息框，并且在用户撤消该消息框时，

BaseThreadStart 调用 ExitProcess，以终止整个进程的运行，而不只是终止线程的运行。

注意，在 BaseThreadStart 函数中，线程要么调用 ExitThread，要么调用 ExitProcess。这意味着线程不能退出该函数，它总是在函数中被撤消。这就是 BaseThreadStart 的原型规定返回 VOID，而它从来不返回的原因。

另外，由于使用 BaseThreadStart，线程函数可以在它完成处理后返回。当 BaseThreadStart 调用线程函数时，它会把返回地址推进堆栈，这样，线程函数就能知道在何处返回。但是，BaseThreadStart 不允许返回。如果它不强制撤消线程，而只是试图返回，那么几乎可以肯定会引发访问违规，因为线程堆栈上不存在返回地址，并且 BaseThreadStart 将试图返回到某个随机内存位置。

4.2.4 一个简单的 Windows 多线程程序

在下面这个程序中，我们在主线程中创建一个副线程，并让两个线程各自显示一条“Hello! World!”字符串。

代码 4.1 一个简单的 Windows 多线程程序

```
#include "windows.h"
#include <iostream>

using namespace std;

DWORD WINAPI ThreadFunc(PVOID pvParam)
{
    cout<<"Created thread says 'Hello!World!'"<<endl;
    return 0;
}

int main()
{
    HANDLE ThreadHandle = CreateThread(NULL, 0, ThreadFunc, NULL, 0, NULL);

    //Delay the run of mainthread, because if the mainthread
    //finishes before the created thread does, the process will
    //exit, which may terminate the created thread
    Sleep(100);

    cout<<"Main thread says 'Hello!World!'"<<endl;

    return 0;
}
```

4.3 线程间通信

由于操作系统是随机调度线程的执行的，因此程序员不能预知线程的执行顺序，这虽然可以提高系统资源的利用效率，使程序能够更有效地运行，但也使程序面临了一些随之而来的问题。可以预见，当所有的线程在互相之间不需要进行通信的情况下就能够顺利地进行时，程序的运行性能自然最好，但是，线程很少能够在所有的时间都独立地进行操作。通常在以下两种情况下，线程需要进行相互间的通信：

1、当有多个线程访问共享资源而不希望由于共享使资源遭到破坏时；
2、当一个线程需要将某个任务已经完成的情况通知另外一个或多个线程时。

第一种情况被统称为互斥问题，第二种情况被统称为同步问题，这是线程间通信面对的两个主要问题。下面我们将对其进行分别讨论。

在 Windows 中应用于线程间通信的方法主要包括互锁函数、临界段、事件、互斥量和信号量等方法。

4.3.1 互锁函数

互锁函数是用来解决原子访问问题的一种方式，主要针对操作变量时的原子访问问题。所谓原子访问是指：当线程访问资源时，能够确保没有其它线程同时访问相同的资源。请看下面这个简单的例子：

代码 4.2 互锁函数

```
//Define a global variable.
    Long g_x = 0;

    DWORD WINAPI ThreadFunc1(PVOID pvParam) {
        g_x++;
        return 0;
    }

    DWORD WINAPI ThreadFunc2(PVOID pvParam) {
        g_x++;
        return 0;
    }
```

在这段代码中，声明了一个全局变量 g_x，并将它初始化为 0。现在，假设创建两个线程，一个线程执行 ThreadFunc1，另一个线程执行 ThreadFunc2。这两个函数中的代码是相同的，他们都将 1 添加给全局变量 g_x。因此，当两个线程都停止运行时，你可能希望在 g_x 中看到 2 这个结果。但是在实际情况中，这是不能保证的。下面我们来说明为什么会出现这种情况。假设编译器生成了下面这行代码，以便将 g_x 加 1：

```
MOV EAX, [g_x]          ; Move the value in g_x into a register.
INC EAX             ; Increment the value in the register.
MOV [g_x], EAX          ; Store the new value back in g_x.
```

两个线程不可能在完全相同的时间内执行这个代码。因此，如果一个线程在另一个线程的后面执行这个代码，比如以下面的顺序执行：

```
MOV EAX, [g_x]          ; Thread 1: Move 0 into a register.
INC EAX                 ; Thread 1: Increment the register to 1.
MOV [g_x], EAX          ; Thread 1: Store 1 back in g_x.

MOV EAX, [g_x]          ; Thread 2: Move 1 into a register.
INC EAX                 ; Thread 2: Increment the register to 2.
MOV [g_x], EAX          ; Thread 2: Store 2 back in g_x.
```

这样即可得到期望的结果：g_x 被递增两次成为 2。但是实际情况是，Windows 是个抢占式多线程环境。一个线程可以随时中断运行，而另一个线程则可以随时继续执行。这样，实际发生的情况可能是这样的：

```
MOV EAX, [g_x]          ; Thread 1: Move 0 into a register.
INC EAX                 ; Thread 1: Increment the register to 1.

MOV EAX, [g_x]          ; Thread 2: Move 0 into a register.
INC EAX                 ; Thread 2: Increment the register to 1.
MOV [g_x], EAX          ; Thread 2: Store 1 back in g_x.

MOV [g_x], EAX          ; Thread 1: Store 1 back in g_x.
```

如果代码按这种形式来运行，g_x 中的最后值就不是 2，而是 1 了。这使人感到困惑，因为你对调度程序的控制能力非常小。实际上，如果有 100 个线程在执行这段线程函数，当他们全部退出之后，g_x 的值可能仍然是 1。为了解决这个问题，Windows 提供了一些函数，如果正确使用这些函数，就能确保产生期望的结果。

为了解决上面的问题，我们需要一种手段来保证值的递增能够以原子操作方式进行，也就是不中断地执行。互锁函数就是一种解决方案。让我们看一看下面这个 InterlockedExchangeAdd 函数：

```
LONG InterlockedExchangeAdd(
    PLONG plAddend,
    LONG lIncrement);
```

这是个最简单的互锁函数。只需调用这个函数，传递一个长变量地址，并指明将这个值递增多少即可。但是这个函数能够保证值的递增以原子操作方式来完成。因此可以将上面的代码

重新编写为下面的形式：

代码 4.3 互锁函数

```
//Define a global variable.
    Long g_x = 0;

    DWORD WINAPI ThreadFunc1(PVOID pvParam) {
        InterlockedExchangeAdd(&g_x, 1);
        return 0;
    }

    DWORD WINAPI ThreadFunc2(PVOID pvParam) {
        InterlockedExchangeAdd(&g_x, 1);
        return 0;
    }
```

通过这个小小的修改，g_x 就能以原子操作方式来递增，因此可以确保 g_x 中的值最后是 2。当然，也可以使用 InterlockedExchangeAdd 减去一个值，只要为第二个参数传递一个负值即可。InterlockedExchangeAdd 将返回在*plAddend 中的原始值。因此，所有的线程都应该设法通过调用互锁函数来修改共享的变量。

下面是另外两个互锁函数：

```
LONG InterlockedExchange(
    PLONG plTarget,
    LONG lValue);

LONG InterlockedExchangePointer(
    PVOID* ppvTarget,
    PVOID pvValue);
```

InterlockedExchange 和 InterlockedExchangePointer 都能够以原子操作方式用第二个参数中传递的值来取代第一个参数中传递的当前值。如果是 32 位应用程序，两个函数都能用另一个 32 位值取代一个 32 位值。但是，如果是个 64 位应用程序，那么 InterlockedExchange 能够取代一个 32 位值，而 InterlockedExchangePointer 则取代 64 位值。两个函数都返回原始值。

还有两个互锁函数：

```
PVOID InterlockedCompareExchange(
    PLONG plDestination,
    LONG lExchange,
    LONG lComparand);
```

```
PVOID InterlockedCompareExchangePointer(
    PVOID* ppvDestination,
    PVOID pvExchange,
    PVOID pvComparand);
```

这两个函数的功能为：首先比较第一个参数所指的值和第三个参数的值，如果相等，则将第一个参数所指的值置为第二个参数，如果不相等则不进行任何操作。如果是 32 位应用程序，那么两个函数都在 32 位值上运行，但是，如果是 64 位应用程序，InterlockedCompareExchange 函数在 32 位值上运行，而 InterlockedCompareExchangePointer 函数则在 64 位值上运行。这两个函数的返回值都是第一个参数所指的原始值。

虽然 Windows 还提供了另外几个互锁函数，但是上面介绍的这些函数就足以实现其他函数能做的功能了。下面是两个其他的函数：

```
LONG InterlockedIncrement(PLONG plAddend);

LONG InterlockedDecrement(PLONG plAddend);
```

这两个函数分别用于将 plAddend 所指的值加 1 和减 1。显然 InterlockedExchangeAdd 函数就能够取代这些较老的函数。

4.3.2 临界段

4.3.2.1 临界段的基本概念

当必须以原子操作方式来修改单个值时，互锁函数家族是相当有用的，但是大多数实际工作中的编程问题要解决的是比单个 32 位或 64 位值复杂得多的数据结构。为了以原子操作方式使用更加复杂的数据结构，必须使用 Windows 提供的其它的某些特性，临界段就是其中之一。

临界段也称作关键代码段，它是指一个小代码段，在它能够执行前，它必须独占对某些共享资源的访问权。一旦线程执行进入了临界段，就意味着它获得了这些共享资源的访问权，那么在该线程处于临界段内的期间，其它同样需要独占这些共享资源的线程就必须等待，直到获得资源的线程离开临界段而释放资源。这是让若干行代码能够以原子操作方式来使用资源的一种方法。当然，系统仍然能够抑制进入临界段的线程的运行，而抢先安排其它不需要独占这些共享资源的线程运行。

4.3.2.2 Win32 API 中的临界段相关函数

有 4 个函数用于临界段。要使用这些函数，你必须定义一个临界段对象，这是一个类型为 CRITICAL_SECTION 的实例，通常为全局变量，这样可以方便多个线程对其的引用，例如：

```
CRITICAL_SECTION cs;
```

这个 CRITICAL_SECTION 数据类型是一个结构，但是其中的字段只能由 Windows 内部使用。这个临界段对象必须首先被程序中的某个线程初始化，通过调用：

```
InitializeCriticalSection(&cs);
```

这样就创建了一个名为 cs 的临界段对象。

当临界段对象被初始化之后，线程可以通过下面的调用进入临界段：

```
EnterCriticalSection(&cs);
```

在这时，线程被认为“拥有”临界段对象。没有两个线程可以同时拥有相同的临界段对象，因此，如果一个线程进入了临界段，那么下一个使用同一个临界段对象调用 EnterCriticalSection 的线程将被迫等待，直到第一个线程通过下面的调用离开临界段：

```
LeaveCriticalSection(&cs);
```

这时，在 EnterCriticalSection 中等待的一个线程将拥有临界段，进而允许继续执行。当临界段不再被程序所需要时，可以通过调用

```
DeleteCriticalSection(&cs);
```

将其删除，该函数释放所有被分配来维护此临界段对象的系统资源。

根据需要，你可以定义多个临界段对象，比如 cs1 和 cs2。例如，如果一个程序有 4 个线程，而前两个线程共享一些数据，那么它们可以使用一个临界段对象，而两外两个线程共享一些其它的数据，那么它们可以使用另一个临界段对象。总的来讲，应该为每个共享资源使用一个 CRITICAL_SECTION 变量。

4.3.2.3 临界段的应用举例

读者可以先运行以下的程序

代码 4.4

```
#include "windows.h"
#include <iostream>

using namespace std;

DWORD WINAPI ThreadFunc1(PVOID pvParam)
{
        for(int i = 1; i <= 100; i++)
        {
            cout<<"Thread1 output: "<<i<<endl;
        }

        return 0;
}
```

```
DWORD WINAPI ThreadFunc2(PVOID pvParam)
{
        for(int i = 1; i <= 100; i++)
        {
            cout<<"Thread2 output: "<<i<<endl;
        }

        return 0;
}

int main()
{
        HANDLE ThreadHandle1 = CreateThread(NULL, 0, ThreadFunc1, NULL, 0, NULL);
        HANDLE ThreadHandle2 = CreateThread(NULL, 0, ThreadFunc2, NULL, 0, NULL);

        //Let the main thread wait for the two running threads
        HANDLE ThreadHandles[2] = {ThreadHandle1, ThreadHandle2};
        WaitForMultipleObjects(2, ThreadHandles, TRUE, INFINITE);

        return 0;
}
```

编译运行该程序会发现，实际的输出字符串十分混乱。这是由于线程共享输出设备时没有协调一致，使资源的利用变得无序造成的。

如果我们为上面的程序段加上临界段，就可以较好的解决这个问题，代码如下：

代码 4.5

```
#include "windows.h"
#include <iostream>

using namespace std;

CRITICAL_SECTION cs;

DWORD WINAPI ThreadFunc1(PVOID pvParam)
{
        for(int i = 1; i <= 100; i++)
        {
            EnterCriticalSection(&cs);
            cout<<"Thread1 output: "<<i<<endl;
            LeaveCriticalSection(&cs);
        }
```

```
        return 0;
}

DWORD WINAPI ThreadFunc2(PVOID pvParam)
{
        for(int i = 1; i <= 100; i++)
        {
            EnterCriticalSection(&cs);
            cout<<"Thread2 output: "<<i<<endl;
            LeaveCriticalSection(&cs);
        }

        return 0;
}

int main()
{
        InitializeCriticalSection(&cs);

        HANDLE ThreadHandle1 = CreateThread(NULL, 0, ThreadFunc1, NULL, 0, NULL);
        HANDLE ThreadHandle2 = CreateThread(NULL, 0, ThreadFunc2, NULL, 0, NULL);

        //Let the main thread wait for the two running threads
        HANDLE ThreadHandles[2] = {ThreadHandle1, ThreadHandle2};
        WaitForMultipleObjects(2, ThreadHandles, TRUE, INFINITE);
        DeleteCriticalSection(&cs);

        return 0;
}
```

4.3.3 使用内核对象的线程间通信

前面介绍的互锁函数和临界段都是允许线程保留在用户态来实现线程间通信的机制。用户态通信的优点是它的执行速度非常快，但是它也有其局限性。对于许多应用程序来说，这种机制是不适用的，一个最显著的问题就在于用户态机制只能实现同一进程内线程的通信。下面我们将介绍几种使用内核对象来实现线程间通信的机制。你将会看到，内核对象机制的适应性远远优于用户态机制。实际上，内核对象机制的唯一不足之处就是它的速度比较慢。当调用后面将要介绍的新函数时，调用线程必须从用户态切换为内核态。这个切换需要很大的代价：往返一次需要占用 x86 平台上的大约 1000 个 CPU 周期，当然，这还不包括执行函数代码的时间。

Windows 中的某些内核对象包含着一个特殊的属性，用来表示该内核对象是处于已通知状态

还是未通知状态，在介绍如何使用这个属性来实现线程间通信之前，我们首先列出拥有这一属性的内核对象，它们包括：

- ◆进程
- ◆线程
- ◆作业
- ◆文件
- ◆控制台输入
- ◆文件修改通知
- ◆事件
- ◆可等待定时器
- ◆信号量
- ◆互斥量

不同的内核对象是否处于已通知状态或未通知状态的含义是不同的。以进程或线程来说，当进程正在运行时，进程内核对象处于未通知状态，当进程中止运行的时候，它就变为已通知状态。另外，用于控制每个对象的已通知/未通知状态的规则要根据对象的类型而定，比如对于进程或线程而言，这种状态的改变是不可由应用程序设置的，它是由操作系统根据进程或线程的运行状态自动设置的，而其它某些内核对象的状态则是可以由应用程序设置的。已通知/未通知状态的改变是通过改变内核对象中的一个布尔值完成的，未通知状态对应为FALSE，已通知状态对应为 TRUE。

线程可以使自己进入等待状态，直到一个对象变为已通知状态，这一功能是通过等待函数完成的。线程调用等待函数，即可使自己进入等待状态，如果它等待的内核对象正处于未通知状态，而一旦这个内核对象变为已通知状态，线程即可立即恢复就绪状态。等待函数中最常用的是 WaitForSingleObject：

```
DWORD WaitForSingleObject(
    HANDLE hObject,
    DWORD dwMilliseconds);
```

当线程调用该函数时，第一个参数 hObject 表示一个能够支持已通知/未通知状态的内核对象，第二个参数 dwMilliseconds 允许该线程指明，为了等待该对象变为已通知状态，它将等待多长时间。

调用下面这个函数将告诉系统，调用函数准备等待到 hProcess 句柄标识的进城中止运行为止：

```
WaitForSingleObject(hProcess, INFINITE);
```

INFINITE 参数告诉系统，调用线程愿意永远等待下去，直到该进程中止运行。

WaitForSingleObject 函数的返回值能够指明调用线程为什么再次变为可调度状态。如果线程等待的对象变为已通知状态，那么返回值是 WAIT_OBJECT_0。如果设置的超时已经到期，则返回值是 WAIT_TIMEOUT。如果将一个错误的值（如一个无效句柄）传递给 WaitForSingleObject，那么返回值将是 WAIT_FAILED。

下面这个函数 WaitForMultipleObjects 与 WaitForSingleObject 函数很相似，区别在于它允许调用线程同时查看若干个内核对象的已通知状态：

```
DWORD WaitForMultipleObjects(
    DWORD dwCount,
    CONST HANDLE* phObjects,
    BOOL fWaitAll,
    DWORD dwMilliseconds);
```

dwCount 参数用于指明想要让函数查看的内核对象的数量。这个值必须在 1 与 MAXIMUM_WAIT_OBJECTS（在 Windows 头文件中定义为 64）之间。phObjects 参数是指向内核对象句柄的数组的指针。fWaitAll 参数是一个布尔值，如果为 TRUE，那么它告诉该函数只有当指定的所有内核对象都变为已通知状态函数才返回，如果为 FALSE，那么它告诉该函数只要指定的内核对象中有一个内核对象变为已通知状态函数即返回。dwMilliseconds 参数的作用与它在 WaitForSingleObject 中的作用完全相同。

WaitForMultipleObjects 函数的返回值与 WaitForSingleObject 大致相同，只是当发 fWaitAll 参数为 FALSE 时，应用程序可能希望知道是哪个对象变为已通知状态，这时的返回值为 WAIT_OBJECT_0 与 WAIT_OBJECT_ 0 +dwCount-1 之间的一个值。

4.3.4 事件

4.3.4.1 事件的基本概念

在所有内核对象中，事件内核对象是个最简单的对象。它包括一个使用计数，一个用于指明该事件是个自动重置的事件还是一个人工重置的事件的布尔值，还有一个用于指明该事件处于已通知状态还是未通知状态的布尔值。它主要用于标识一个操作是否已经完成。有两种不同类型的事件对象，一种是人工重置的事件，另一种是自动重置的事件。当人工重置的事件得到通知时，等待该事件的所有线程均变为可调度线程。当一个自动重置的事件得到通知时，等待该事件的线程中只有一个线程变为可调度线程。

4.3.4.2 Win32 API 中的事件相关函数

下面是 CreateEvent 函数，用于创建事件内核对象：

```
HANDLE CreateEvent(
    PSECURITY_ATTRIBUTES psa,
    BOOL fManualReset,
    BOOL fInitialState,
    PCTSTR pszName);
```

psa 是设置内核对象安全性的参数，在此不再重述。FManualReset 参数是个布尔值，它能够告诉系统是创建一个人工重置的事件（TRUE）还是创建一个自动重置的事件（FALSE）。fInitialState 参数 用于指明该事件是要初始化为已通知状态（TRUE）还是未通知状态（FALSE）。当系统创建事件对象后，CreateEvent 就将与进程相关的句柄返回给事件对象。其它进程中的线程可以获得对该对象的访问权，方法有四种：使用在 pszName 参数中传递的相同值，使用继承性，使用 DuplicateHandle 函数等来调用 CreateEvent，和调用 OpenEvent。

OpenEvent 的形式为：

```
HANDLE OpenEvent(
    DWORD fdwAccess,
    BOOL fInherit,
    PCTSTR pszName);
```

其中，fdwAccess 标识对事件的访问方式，通常设置为 EVENT_ALL_ACCESS。fInherit 标识该事件的继承属性，如果为 TRUE，那么调用 OpenEvent 的进程的子进程将可以继承该事件，如果为 FALSE 则不可继承。pszName 与 CreateEvent 中的相同，OpenEvent 根据这个参数来引用事件。

一旦事件已经创建，就可以直接控制它的状态。当调用 SetEvent 时，可以将时间改为已通知状态：

```
BOOL SetEvent(HANDLE hEvent);
```

当调用 ResetEvent 函数时，可以将该事件改为未通知状态：

```
BOOL ResetEvent(HANDLE hEvent);
```

事件在创建时可以指定是人工重置的还是自动重置的。人工重置就是指通过 SetEvent 和 ResetEvent 来设置事件的通知状态，而自动重置的含义则是：当事件被 SetEvent 函数设置为已通知状态时，在等待事件的线程中，只有一个线程被恢复，之后系统自动将事件置为未通知状态。在这个过程中，哪个线程被恢复是不可预知的。

4.3.4.3 事件的应用举例

事件的主要用途是标志事件的发生，并以此协调线程的执行顺序。下面是一个简单的应用事件的例子，用户在主线程中输入命令，控制新建线程的运行：

代码 4.6

```
#include "windows.h"
#include <iostream>
#include <string>

using namespace std;

CRITICAL_SECTION cs;

DWORD WINAPI ThreadFunc(PVOID pvParam)
{
        EnterCriticalSection(&cs);
        cout<<"Created thread: Created Thread is started."<<endl;
```

```
        cout<<"Created thread: Created Thread is waiting continue
command..."<<endl;
        LeaveCriticalSection(&cs);

        //Get the handle of the event
        HANDLE phEvent = OpenEvent(EVENT_ALL_ACCESS, TRUE, "ContinueCommand");

        //Waiting the event to be set
        WaitForSingleObject(phEvent, INFINITE);

        cout<<"Created thread: Recieved continue command."<<endl;
        cout<<"Created thread: Thread runs again."<<endl;
        Sleep(2000);
        cout<<"Created thread: Thread finished."<<endl;

        return 0;
}

int main()
{
        InitializeCriticalSection(&cs);

        //Create an Event
        HANDLE hEvent = CreateEvent(NULL, FALSE, FALSE, "ContinueCommand");

        cout<<"Main thread: Creating new thread..."<<endl;
        HANDLE ThreadHandle = CreateThread(NULL, 0, ThreadFunc, NULL,
CREATE_SUSPENDED, NULL);
        cout<<"Main thread: New thread created."<<endl;
        ResumeThread(ThreadHandle);


        string input;
        while(TRUE)
        {
            EnterCriticalSection(&cs);
            cout<<"Main thread: input command, please."<<endl;
            LeaveCriticalSection(&cs);
            cout<<">";
            cin>>input;
            if(input == "continue")
            {
                cout<<"Main thread: OK, let the thread continue to run."<<endl;
```

```
            //Set the Event
            SetEvent(hEvent);

            break;
        }
    }

    //Waiting the created thread to finish
    WaitForSingleObject(ThreadHandle, INFINITE);

    cout<<"Main thread: Created thread finished, I see."<<endl;

    DeleteCriticalSection(&cs);

    //Close the event
    CloseHandle(hEvent);

    return 0;
}
```

4.3.5 互斥量

4.3.5.1 互斥量的基本概念

互斥量（mutex）是一种内核对象，它能够确保线程拥有对单个资源的互斥访问权。互斥量包含一个使用数量，一个线程 ID 和一个递归计数器。互斥量的行为特性与临界段相同，但是互斥量属于内核对象，而临界段属于用户方式对象。这意味着互斥量的运行速度比临界段要慢，但是这也意味着不同进程中的多个线程能够访问单个互斥量，并且这意味着对等待访问资源的线程可以设定一个超时值。

互斥量中的线程 ID 用于标识系统中的哪个线程当前拥有互斥对象，如果为 0 则表示没有线程拥有该互斥量，递归计数器用于指明该线程拥有互斥对象的次数。互斥对象有许多用途，属于最常用的内核对象之一，经常被用于保护由多个线程访问的内存块。如果多个线程要同时访问内存块，内存块中的数据就可能遭到破坏。互斥对象能够保证访问内存块的任何线程拥有对该内存快的独占访问权，这样就能够保证数据的完整性。

4.3.5.2 Win32 API 中的互斥量相关函数

若要使用互斥对象，必须有一个进程首先调用 CreateMutex，以便创建互斥对象：

```
HANDLE CreateMutex(
PSECURITY_ATTRIBUTES psa,
BOOL fInitialOwner,
```

```
    PCTSTR pszName);
```

psa 和 pszName 参数与 CreateEvent 中的含义相同。fInitialOwner 用于控制互斥对象的初始状态。如果传递 FALSE（这是通常状况下的做法），那么互斥量中的线程 ID 被置为 0，互斥量处于已通知状态；如果传递 TRUE，那么该互斥量的线程 ID 被设置为调用 CreateMutex 的线程的 ID，互斥量也处于未通知状态。

通过调用 OpenMutex，另一个进程可以获得与本进程相关的互斥量对象的句柄：

```
HANDLE OpenMutex(
    DWORD fdwAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);
```

参照 OpenEvent 函数，OpenMutex 函数的参数是很容易理解的。

线程是通过 WaitForSingleObject 函数或 WaitForMultipleObject 函数来等待互斥量的。如果线程成功获得了互斥量，那么互斥量中的线程 ID 将记录该线程的 ID，并将递归计数加 1。线程可以多次等待互斥量，递归计数则记录线程等待到互斥量的次数。而当线程不再需要共享资源的独占访问权时，它必须调用 ReleaseMutex 函数来释放互斥量：

```
BOOL ReleaseMutex(HANDLE hMutex);
```

获得互斥量的线程调用 ReleaseMute 函数使互斥量中的递归计数减 1。如果线程多次等待到了互斥量，那么只有当它调用对应次数的 ReleaseMutex，使递归计数为 0 时，互斥量才会将其中的线程 ID 设为 0，并恢复已通知状态。

互斥量不同于所有其它内核对象，因为互斥对象有一个“线程所有权”的概念，这使对互斥量的使用有不同的规则。在释放互斥量时，就会遇到两种特殊的情况，一种是：如果调用 ReleaseMutex 的线程并没有获得互斥量，也就是说该线程的 ID 不等于互斥量中的线程 ID，那么 ReleaseMutex 函数将不进行任何操作，而是将 FALSE 返回给调用线程。此时调用 GetLastError，将返回 Error_NOT_OWNER。另一种情况是：如果在释放互斥量之前，拥有互斥量的线程中止运行，那么系统将自动把互斥量中的线程 ID 和递归计数复置为 0，使互斥量处于已通知状态。

4.3.5.4 互斥量的应用举例：

下面这个例子与前面临界段的例子大致相同，只是用互斥量完成相同的工作。读者可以看出，互斥量与临界段的功能十分相似。这次我们使用 pvParam 参数为线程传递互斥量对象的句柄。

代码 4.7

```
#include "windows.h"
#include <iostream>

using namespace std;
```

```
DWORD WINAPI ThreadFunc1(PVOID pvParam)
{
        HANDLE* phMutex = (HANDLE*)pvParam;

        for(int i = 1; i <= 100; i++)
        {
            WaitForSingleObject(*phMutex, INFINITE);
            cout<<"Thread1 output: "<<i<<endl;
            ReleaseMutex(*phMutex);
        }

        return 0;
}

DWORD WINAPI ThreadFunc2(PVOID pvParam)
{
        HANDLE* phMutex = (HANDLE*)pvParam;

        for(int i = 1; i <= 100; i++)
        {
            WaitForSingleObject(*phMutex, INFINITE);
            cout    <<"Thread2 output: "<<i<<endl;
            ReleaseMutex(*phMutex);
        }

        return 0;
}

int main()
{
        HANDLE hMutex = CreateMutex(NULL, FALSE, "DisplayMutex");

        HANDLE ThreadHandle1 = CreateThread(NULL, 0, ThreadFunc1, &hMutex, 0,
NULL);
        HANDLE ThreadHandle2 = CreateThread(NULL, 0, ThreadFunc2, &hMutex, 0,
NULL);

        //Let the main thread wait for the running two thread
        HANDLE ThreadHandles[2] = {ThreadHandle1, ThreadHandle2};
        WaitForMultipleObjects(2, ThreadHandles, TRUE, INFINITE);
        CloseHandle(hMutex);

        return 0;
```

}

4.3.6 信号量

4.3.6.1 信号量的基本概念

信号量也是一种内核对象，用于对资源进行计数，它除了像其它内核对象一样拥有使用计数外，还拥有两个带符号的 32 位整数值，一个是最大资源数量，用于标识信号量能够控制的资源的最大数量，另一个是当前资源数量，用于标识当前可以使用的资源数量。

信号量的使用规则是这样的，如果当前资源数量大于 0，那么等待信号量的线程可以获得一个资源并继续执行，信号量的当前资源数将减 1；如果当前资源数为 0，那么等待信号量的线程将处于等待状态，直到有线程释放信号量，使当前资源数大于 0。当前资源数不会超过最大资源数量的值，也不会小于 0。

4.3.6.2 Win32 API 中的信号量相关函数

下面的函数用于创建信号量内核对象：

```
HANDLE CreateSemaphore(
    PSECURITY_ATTRIBUTE psa,
    LONG lInitialCount,
    LONG lMaximumCount,
    PCTSTR pszName);
```

psa 和 pszName 两个参数不难理解，lInitialCount 用于标识信号量的初始资源数，lMaximumCount 用于标识信号量的最大资源数。

通过调用 OpenSemaphore 函数，另一个进程可以获得它自己的进程与现有信号量的相关句柄：

```
HANDLE OpenSemaphore(
    DWORD fdwAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);
```

调用调用 ReleaseSemaphore 函数，线程就能够对信号量的当前资源数量进行递增：

```
BOOL ReleaseSemaphore(
    HANDLE hsem,
    LONG lReleaseCount,
    PLONG plPreviousCount);
```

该函数只是将 lReleaseCount 的值添加到当前资源数量中，通常情况下传递的是 1，当然也可以传递更大的值。plPreviousCount 用来返回信号量的原先的当前资源数量，通常几乎没

有应用程序关心这个值，因此可以传递 NULL 将它忽略。

4.3.6.3 信号量的应用举例

比如两个线程中分别有一个初值为 0 的 int 型局部变量，两个线程的行为就是在一个循环中使自己的这个整形变量递增，但有一个约束：在递增的过程中，这两个值的差不能超过 5。解决这个问题就可以使用信号量。大致的解决方案可以是这样的：

代码 4.8

```
hsem1 = CreateSemaphore(NULL, 5, 10, “sem1”);
hsem2 = CreateSemaphore(NULL, 5, 10, “sem2”);

DWORD WINAPI ThreadFunc1(PVOID pvParam){
        int i1 = 0;
        for(int i = 0; i < 10000; i++)
        {
            WaitForSingleObject(hsem1, INFINITE);
            ReleaseSemaphore(hsem2, 1, NULL);
            i1++;
    }
        return(i1);
}

DWORD WINAPI ThreadFunc2(PVOID pvParam){
        int i2 = 0;
        for(int i = 0; i < 10000; i++)
        {
            WaitForSingleObject(hsem2, INFINITE);
            ReleaseSemaphore(hsem1, 1, NULL);
            i2++;
    }
        return(i2);
}
```

4.4 调度优先级

Windows 是一个多任务多线程操作系统，它采取的调度方法是基于优先级的时间片轮转抢占式调度。每个线程都会被赋予一个从 0（最低）到 31（最高）的优先级号码。相对高优先级的线程总是被系统优先调度;对于同优先级的线程，系统将对其采取时间片轮转的调度方式;对于相对低优先级的线程，则将处于就绪状态，直到相对高优先级的线程执行完毕。并且，高优先级的线程可以即时地获得 CPU，不管正在运行的低优先级线程在做什么，都将被暂停而让出 CPU。

Windows 对于进程支持 6 个相对的优先级类，即：空闲、低于正常、正常、高于正常、高和实时。对于线程支持 7 个相对的优先级类，即：空闲、最低、低于正常、正常、高于正常、最高和关键时间。下面两个表格分别说明了这两个优先级类的含义：

进程优先级类	描述
实时	进程中的线程必须立即对事件作出响应，以便执行关键时间的任务。该进程中的线程还会抢先于操作系统组件之前运行。使用本优先级类时必须非常小心
高	进程中的线程必须立即对事件作出响应，以便执行关键时间的任务。Task Manager 就是在这个类上运行的，以便用户可以撤销脱离控制的进程
高于正常	进程中的线程在正常优先级与高优先级之间运行
正常	进程中的线程没有特殊的调度需求
低于正常	进程中的线程在正常优先级与空闲优先级之间运行
空闲	进程中的线程在系统空闲时运行。该优先级通常由屏幕保护程序或后台实用程序和搜集统计数据的软件使用

表 4.3 进程优先级列表

线程优先级类	描述
关键时间	对于实时优先级类来说，线程在优先级 31 上运行，对于其它优先级类来说，线程在优先级 15 上运行
最高	线程在高于正常优先级的上两级上运行
高于正常	线程在正常优先级的上一级上运行
正常	线程在进程的优先级类上正常运行
低于正常	线程在低于正常优先级的下一级上运行
最低	线程在低于正常优先级的下两级上运行
空闲	对于实时优先级类来说，线程在优先级 16 上运行，对于其它优先级类来说，线程在优先级 1 上运行

表 4.4 线程优先级列表

应用程序开发人员从来不必具体设置线程的从 0 到 31 的优先级，这个具体优先级是由线程所在进程的优先级类和线程本身的优先级类映射决定的。具体的映射方式是随着系统的版本不同而可以有所不同的，这为系统版本的升级提供了灵活性。

下面的表格列出了在 Windows 2000 中的映射关系：

线程 \ 进程	空闲	低于正常	正常	高于正常	高	实时

关键时间	15	15	5	15	15	31
最高	6	8	10	12	15	26
高于正常	5	7	9	11	14	25
正常	4	6	8	10	13	24
低于正常	3	5	7	9	12	23
最低	2	4	6	8	11	12
空闲	1	1	1	1	1	16

表 4.5 优先级映射关系

注意，上表中并没有优先级等级为 0 的线程，这是因为 0 优先级保留给零页线程使用，系统不允许任何其它线程拥有 0 优先级。所谓零页线程，是系统引导时创建的一个特殊的线程，它的优先级为 0，当系统中没有任何线程需要执行操作时，零页线程负责将系统中的所有空闲 RAM 页面置 0。另外，下列优先级等级是无法使用的：17、18、19、20、21、27、28、29 和 30。如果编写一个以内核方式运行的设备驱动程序，可以获得这些优先级等级，而用户方式的应用程序则不能。

设定程序优先级的 API：
对于进程而言，当调用 CreateProcess 时，可以在 fdwCreate 参数中传递需要的优先级类。下表显示了优先级类的宏标识符：

进程优先级	宏标识符
Real Time	REALTIME_PRIORITY_CLASS
High	HIGH_PRIORITY_CLASS
Above-normal	ABOVE_NORMAL_PRIORITY_CLASS
Normal	NORMAL_PRIORITY_CLASS
Below-normal	BELOW_NORMAL_PRIORITY_CLASS
Idle	IDLE_PRIORITY_CLASS

表 4.6 进程优先级类的宏标识符

也可以通过调用 SetPriorityClass 函数来改变指定进程的优先级类：

```
BOOL SetPriorityClass(
    HANDLE hProcess,
    DWORD fdwPriority);
```

该函数将 hProcess 标识的进程的优先级类改为 fdwPriority 参数中的设定值。

对于线程而言，在它创建时，它的优先级总是正常优先级，CreateThread 函数并没有为调用者提供一个设置新线程优先级的参数。若要设置线程的优先级，必须调用下面的函数：

```
BOOL SetThreadPriority(
    HANDLE hThread,
    Int nPriority);
```

其中，hThread 参数用于标识想要改变优先级的单个线程，nPriority 参数是下表中的 7 个

宏标识符之一：

线程优先级类	宏标识符
关键时间	THREAD_PRIORITY_TIME_CRITICAL
最高	THREAD_PRIORITY_HIGHEST
高于正常	THREAD_PRIORITY_ABOVE_NORMAL
正常	THREAD_PRIORITY_NORMAL
低于正常	THREAD_PRIORITY_BELOW_NORMAL
最低	THREAD_PRIORITY_LOWEST
空闲	THREAD_PRIORITY_IDLE

表 4.7 线程优先级的宏标识符

4.5 线程池

在某些应用程序中，为执行某些任务，开发人员可能需要动态的分配一些线程。完成任务所需的这些线程在数量上完全不受开发人员的控制，可能非常多，也可能仅需要几个，各种情况的差别可能很大。例如，一个 Web 服务器应用程序，有时会因为服务器空闲而不需要做任何处理工作，有时却需要在给定的时间内处理多达数千个请求。从软件角度处理这种情况的一个方法就是采用动态线程创建技术。当系统开始不断接收越来越多的工作时，开发人员需要不断创建新的线程来处理输入的请求。而当系统变得空闲时，由于没有足够多的工作需要完成，并且线程所占用的系统资源也非常宝贵，所以开发人员可能需要中止那些在系统处于最大负载状况时所创建的一些线程。

动态线程创建技术中存在着一些问题。首先，线程的创建是一个开销很大的操作。当初于峰值流量时，Web 服务器在线程创建上所花费的时间可能比实际响应用户请求的时间还要多。为了解决这个问题，开发人员可以在应用程序启动时就创建一组线程，当有处理请求来到时，这些线程已经做好了准备，这样就可以解决线程创建所带来的开销问题了。但是，这里依然还存在着其他的问题：创建多少线程最合适？如何根据当前系统的负载状况对这些线程进行最优调度？在应用层，这些参数对大多数开发人员来说是不可见的，因此，操作系统对线程池技术提供的一些支持就显得非常有意义了。

从 Windows 2000 起，Microsoft 开始提供线程池 API，该 API 可以极大地减少开发人员实现线程池所需完成的代码量。使用线程池最重要的函数是 QueueUserWorkItem：

```
BOOL QueueUserWorkItem(
    LPTHREAD_START_ROUTINE Function,
    PVOID Context,
    ULONG Flags);
```

Function 参数是一个函数指针，指向线程池中的线程必须要完成的工作。该函数必须具有以下形式：

```
DWORD WINAPI Function(LPVOID parameter);
```

读者可以发现，这个函数与创建线程时的线程函数形式相同。这个函数的返回值就是线程的退出代码，该退出代码可以通过调用 GetExitCodeThread()获得。Context 参数是一个 void 指针，与传递给线程函数的 pvParam 参数功能相同。Flags 参数将在后面进行介绍。

当第一次调用 QueueUserWorkItem 时，Windows 将创建一个线程池，其中的一个线程将执行 Function 函数，函数执行完成后，该线程返回线程池，等待新的任务。由于 Windows 依赖于该过程来完成线程池的功能，因此 Function 中不能有进行任何中止该线程的调用，如 ExitThread。假如当调用 QueueUserWorkItem 时，没有可用的线程，Windows 就可以通过创建额外的线程增加线程池中线程的数量。线程池中的线程的数量是动态的，并且受 Windows 的控制，Windows 内部的调度算法决定处理当前线程工作负载的最佳方式。

如果知道所要处理的工作需要很长时间才能完成，可以在调用 QueueUserWorkItem 时，将参数 Flags 设置为 WT_EXECUTELONGFUNCTION。这时如果线程池中所有的线程都处于忙状态，那么 Windows 将自动创建新的线程。

Windows 线程池中的线程有两种类型：一种可以用来处理异步 I/O，另一种则不能。前者依赖于 I/O 完成端口。I/O 完成端口(I/O completion port)是一种 Windows 内核对象，它可以将线程和 I/O 端口绑定在特定的系统资源上。对带有完成端口的 I/O 进行处理是一个复杂的过程，在此不多介绍。调用 QueueUserWorkItem 时，需要标识哪些线程执行 I/O，哪些线程不执行 I/O。将 QueueUserWorkItem 中的 Flags 设置成 WT_EXECUTIONDEFAULT，就可以告知线程池该线程不执行异步 I/O，从而可以对其进行相应的管理。对于执行异步 I/O 的线程，则应该将其 Flags 设置为 WT_EXECUTEINIOTHREAD。

当使用多个线程对目标工作进行功能分解时，也可以考虑使用线程池 API 来减轻程序设计的负担，使 Windows 有机会协助进行线程的管理，从而使应用程序能够达到最佳的性能。

第五章 OpenMP 多线程编程及性能优化

本书前面章节已经介绍了在 Windows/Unix/Linux 操作系统下使用标准线程库及接口进行多线程编程的基本方法与技术。在此基础上，本章将进一步介绍 OpenMP，一种简单的编写多线程应用程序的方法，而无需程序员手工处理复杂的线程创建、同步、负载平衡和销毁等技术细节。

OpenMP 是一种针对共享内存的多线程编程技术，由一些具有国际影响力的大规模软件和硬件厂商共同定义标准（www.openmp.org）。OpenMP 目前的规范（Specification）版本是 2.5，支持 Fortran，C 和 C++。它是一种通过提供平台无关的编译制导（pragmas）、函数调用和环境变量的方式，显式地制导编译器如何以及何时利用应用程序中的并行性。这一章将给出一些示例来说明这一点。程序开发人员无需关心那些实现细节，这是编译器和 OpenMP 线程库的工作。程序开发人员只需要认真考虑哪些循环应该以多线程方式执行，以及如何重构算法以便在多核处理器上获得更好的性能等问题。

5.1 OpenMP 编程简介

5.1.1 OpenMP 多线程编程发展概况

OpenMP 是一种面向共享内存以及分布式共享内存的多处理器多线程并行编程语言。OpenMP 是一种能够被用于显式制导多线程、共享内存并行的应用程序编程接口（API）。OpenMP 的规范由 SGI 发起，由一组主要的计算机硬件和软件厂商共同制定并认可，希望以后能够成为美国国家标准（ANSI standard）。

OpenMP 标准诞生于 1997 年，目前其结构审议委员会（Architecture Review Board, ARB）正在制定并即将推出 OpenMP 3.0 版本。可以从 www.openmp.org 官方网站上看到有关于 OpenMP 最新的发展，并且能够下载到关于 OpenMP 的最新标准。最早的标准发表于 1997 年 10 月，为 Fortran 1.0 标准。标准版本 2.5 发表于 2005 年 5 月，支持语言 Fortran77，Fortran90，Fortran95 以及 C/C++；同时在平台支持上，OpenMP 能够支持多种平台，包括大多数的类 UNIX 系统以及 Windows NT 系统（Windows 2000，Windows XP，Windows Vista 等）。

OpenMP 最初是为共享内存的多处理器系统设计的并行编程方法，这与通过消息传递进行并行编程的模型有很大的不同。其体系结构如下图所示。

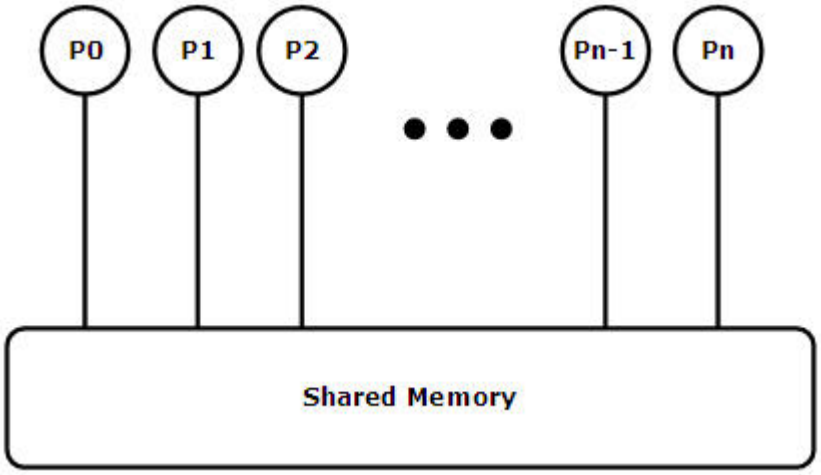

图 5. 1 共享内存多处理器体系结构

从图中可以看出，所有的处理器都被连接到一个共享的内存单元上，处理器在访问内存的时候使用的是相同的内存地址空间。由于内存是共享的，因此，某一个处理器写入内存的数据会立刻被其它处理器访问到。常见的对称多处理（Symmetrical Multi-Processing，SMP）就是一种共享内存的体系结构，即在一个计算机上汇集了一组处理器，各个处理器之间共享内存子系统以及总线结构。除此之外，分布式共享内存的系统也属于共享内存多处理器结构，分布式共享内存将多机的内存资源通过虚拟化的方式形成一个统一的内存空间提供给多机上的处理器使用，OpenMP 对这样的机器也提供一定的支持。

与共享内存方式相对的就是分布式内存，每一个处理器或者一组处理器会有一个自己私有的内存单元，共享或者不共享一个公用的内存单元。此时，程序员需要关心数据的实际存放问题，因为数据存放未知会直接影响相应处理器的处理速度。通过网络连接的个人计算机系统组成的集群以及工作站网络（NOWs：Network of Workstations）就是典型的分布式内存多处理器系统。这种系统一般使用特定的消息传递库，例如 MPI 或者并行虚拟机（PVM）来进行编程，OpenMP 不专注于这种编程模式。OpenMP3.0 的标准开始支持分布式结构的集群，英特尔也有 ClusterOpenMP 对此种方式进行支持。但是从设计初衷来说，OpenMP 并不适合于这种编程模式。

5.1.2 OpenMP 多线程编程基础

OpenMP 的编程模型以线程为基础，通过编译制导语句来显示地制导并行化，为编程人员提供了对并行化的完整控制。在动手编写 OpenMP 程序之前，我们需要一些有关 OpenMP 多线程编程的基础知识。本小节将讲解 OpenMP 程序的执行模型，OpenMP 程序中所涉及的编译制导语句与库函数以及 OpenMP 程序执行环境的初步介绍，使得初学者能够有一个关于 OpenMP 程序的总体印象。

OpenMP 的执行模型采用 Fork-Join 的形式，如果读者对多线程编程有一定的经验的话，对这种执行模式应该非常熟悉。Fork-Join 执行模式在开始执行的时候，只有一个叫做主线程的运行线程存在。主线程在运行过程中，当遇到需要进行并行计算的时候，派生出（Fork，创建新线程或者从线程池中唤醒已有线程）线程来执行并行任务。在并行执行的时候，主线

程和派生线程共同工作。在并行代码结束执行后，派生线程退出或者挂起，不再工作，控制流程回到单独的主线程中（Join，即多线程的会和）。

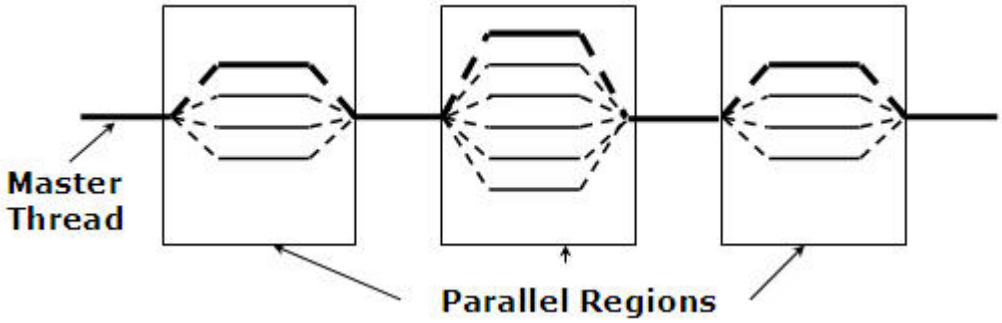

图 5. 2 OpenMP 运行时的 Fork-Join 模型

上图是共享内存多线程应用程序的 Fork-Join 模型，从图中我们可以直观地看出多线程应用程序 Fork-Join 的执行模型是如何运行的。主线程在运行的过程中，遇到并行编译制导语句，根据环境变量、根据实际需要（比如循环的迭代次数）派生出若干个线程。此时，次线程与主线程同时运行。在运行的过程中，某一个派生线程遇到了另外一个并行编译制导语句，派生出了另外一组线程。新的线程组共同完成一项任务，但对于原有的线程来说，新线程组的工作类似于一块串行程序，对原有的线程不会产生影响。新线程组在通过一个隐含的同步屏障后（barrier），汇合（join）成原有的线程。最后，原有的线程组汇合成主线程，执行完最后的程序代码并退出。

OpenMP 同时支持 C/C++语言和 Fortran 语言，可以选择任意一种语言以及支持 OpenMP 的编译器编写 OpenMP 程序。OpenMP 在两种语言上的语法类似，但有一些细小的差别，这主要是由于语言本身的性质所决定的。总体上来说，OpenMP 的功能由两种形式提供：编译制导语句与运行时库函数，并通过环境变量的方式灵活控制程序的运行，例如通过 OMP_NUM_THREADS 值来控制运行的线程的数目。而这其中，编译制导语句是 OpenMP 的精髓，提供了将一个串行程序渐进的改造为并行程序的能力，而对于不支持 OpenMP 编译制导语句的编译器，这些编译制导语句又可以被忽略，完全和原来的串行程序兼容。对于运行时库函数，则只有在必须的情况下才考虑调用。一般不推荐过多的使用 OpenMP 的运行时库函数，因为这会破坏 OpenMP 程序与原有串行程序的兼容性。

编译制导语句

编译制导语句的含义是在编译器编译程序的时候，会识别特定的注释，而这些特定的注释就包含着 OpenMP 程序的一些语义。例如在 C/C++程序中，用#pragma omp parallel 来标识一段并行程序块。在一个无法识别 OpenMP 语意的普通编译器中，这些特定的注释会被当作普通的注释而被忽略。因此，如果仅仅使用编译制导语句，编写完成的 OpenMP 程序就能够同时被普通编译器与支持 OpenMP 的编译器处理。这种性质带来的好处就是用户可以用同一份代码来编写串行或者并行程序，或者在把串行程序改编成并行程序的时候，保持串行源代码部分不变，极大地方便了程序编写人员。在 C/C++程序中，OpenMP 的所有编译制导语句以 #pragma omp 开始，后面跟具体的功能指令。即具有如下的形式：

#pragma omp <directive> [clause[[,] clause]…]

其中 directive 部分就包含了具体的编译制导语句，包括 parallel, for, parallel for, section, sections, single, master, critical, flush, ordered 和 atomic。这些编译制导语句或者用来工作共享，或者用来同步，本章将讨论大部分的编译制导语句。后面的可选子句 clause 给出了相应的编译制导语句的参数，子句可以影响到编译制导语句的具体行为，每一个编译制导语句都有一系列适合它的子句，其中有五个编译制导语句（master, critical, flush, ordered, atomic）不能跟相应的子句。

在 Fortran 中，对应的 OpenMP 注释的形式为

```
C$OMP <directive> [clause[ [,] clause]…]
*$OMP <directive> [clause[ [,] clause]…]
!$OMP <directive> [clause[ [,] clause]…]
```

但必须要注意的是，这样的注释必须从行的第 1～5 列开始，第 6 列元素必须为空，或者是+表示该行是上一行的继续。

运行时库函数

另外一种提供 OpenMP 功能的形式就是 OpenMP 的运行时库函数，OpenMP 运行时函数库原本用以设置和获取执行环境相关的信息，它们当中也包含一系列用以同步的 API。要使用运行时函数库所包含的函数，必须在相应的源文件中包含 OpenMP 头文件即 omp.h。如果程序中仅仅使用了编译制导语句，程序可以忽略头文件 omp.h。在下面我们即将看到的函数 omp_get_thread_num()就是一个在运行时函数库中的函数，它用来返回当前线程的标识号。OpenMP 的运行时库函数的使用类同于相应编程语言对外部库函数的调用，因此，如果缺少了相应的运行时库，连接器（Linker）就无法将程序项目正确的连接。这是库函数与编译制导语句不同的方面。

由此我们可以看出，OpenMP 程序同时结合了两种并行编程的方式。通过编译制导语句，我们可以将串行的程序逐步地改造成一个并行程序，达到增量更新程序的目的，减少程序编写人员一定的负担。同时，这种方式也能够将串行程序和并行程序保持在同一个源代码文件当中，减少了维护的负担。但是，由于是编译制导语句，其优势体现在编译的阶段，对于运行阶段则支持较少。因此，OpenMP 也提供了运行时库函数来支持运行时对并行环境的改变和优化，给编程人员足够的灵活性来控制运行时的程序运行状况。但这种方式就打破了源代码在串行和并行之间的一致性。因此，程序员需要同时掌握这两种方法，并在实际工作中灵活应用，以达到快速解决实际问题，保证程序性能的目的。

OpenMP 应用程序在运行的时候，需要运行函数库的支持，并会获取一些环境变量来控制运行的过程。运行函数库在不同的平台上由不同的格式提供支持，例如在 Windows 平台通过动态链接库，而在 Unix 平台上则同时提供动态链接库和静态链接库。环境变量是动态函数库中用来控制函数运行的一些具体参数，例如需要同时派生出多少个线程或者是采用何种多线程调度的模式等。本章接下来的内容会逐步介绍每一个环境变量及其作用。综上所述，构成 OpenMP 应用程序的三个组成部分如下图所示：编译制导语句，运行时函数库以及环境变量。而在这其中，编译制导语句是 OpenMP 组成中最重要的部分，也是编写 OpenMP 程序的关键。

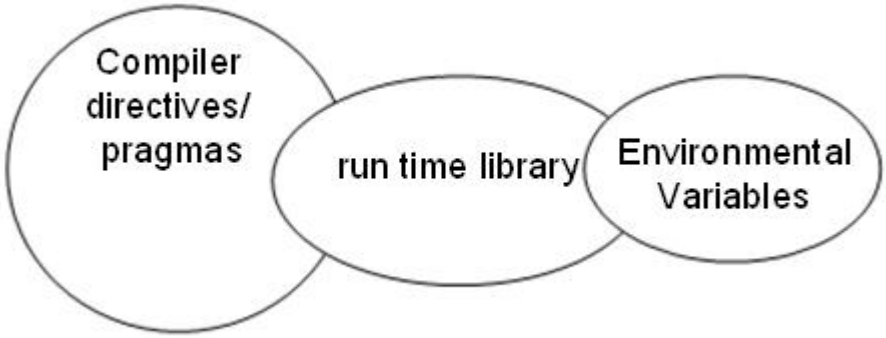

图 5. 3 OpenMP 应用程序的组成部分

5.1.3 编写 OpenMP 程序的准备工作

英特尔 Compiler，无论是 C/C++或 Fortran，都已经增加了对 OpenMP 的支持，因此，本章将主要展示在 Linux 计算平台上，使用英特尔 Compiler 来编写 OpenMP 程序的方法与技术。本章所使用的英特尔编译器版本,无论是 icc(支持 C/C++的编译器)还是 ifort(支持 Fortran 的编译器)，其版本都是 9.0。要使用英特尔编译器来支持 OpenMP 程序，只需在编译时指定“-openmp”选项即可。本章以下例子都将使用 C 语言的 OpenMP 扩展展示例程。

在 OpenMP 中，主要通过对循环或一段结构化代码定义并行区域（Parallel Region）的方式来实现多线程并行。当程序执行到“#pragma omp parallel”时，主线程就并行的派生初若干子线程，如图 5.4 所示。而这些派生出来的子线程在一个区域的结束都会经过一个隐含的栅栏（barrier）而会合。在区域中的所有数据，除非另行说明，否则缺省状况下都是在各线程间共享的。

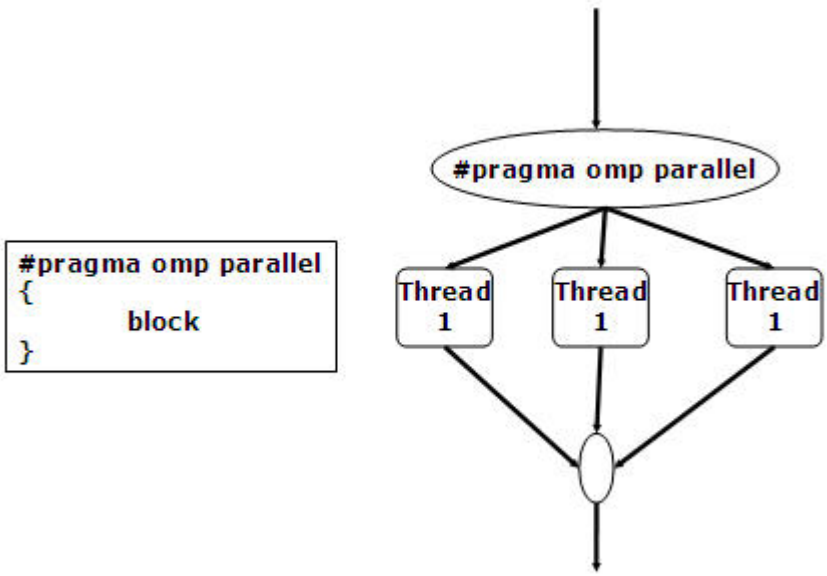

图 5.4 OpenMP 的并行区域

在使用了 OpenMP 编译指示后，程序就开始以多线程的方式执行了，但究竟会派生多少个子

线程完成任务，取决于环境变量 OMP_NUM_THREADS 的设置以及实际任务的需要（比如，循环中一共有多少次迭代）。在 Linux（或者别的类 UNIX 系统中也一样，以后不再赘述）可以使用：

setenv OMP_NUM_THREADS 4

或 export OMP_NUM_THREADS 4

等来设置 OpenMP 中所需要的环境变量。但是 OpenMP 规范中并没有规定这个环境变量的缺省值，因此，许多系统中，缺省状态下，设置 OpenMP 程序的线程数目等于系统中的处理器数目，而英特尔®编译器也沿用了这个规矩。而如果在既有环境变量，又在程序中直接调用 OpenMP 的 API 来设置线程数目的话，系统将根据以下顺序来决定实际的线程数目：

系统缺省值

环境变量

程序中使用 OpenMP 的 API 设置

而这其中，每一条后面的设置将覆盖前一条设置。

正如我们学习所有的编程语言一样，都会以“Hello World”开始我们的学习之旅。下面，我们就来看一下 OpenMP 版本的“Hello World”!

代码 5.1 OpenMP 版本的“Hello World”

```
#include “omp.h”

int main(int argc, char* argv[])
{
  printf(“Hello from serial.\n”);
  printf(“Thread number = %d\n”,omp_get_thread_num());  //串行执行 Sequential execution
  #pragma omp parallel                                  //开始并行执行 Start parallel execution
{
  printf(“Hello from parallel. Thread number=%d\n”,omp_get_thread_num()) ;
}
printf(“Hello from serial again.\n”);
return 0;
}
```

在解释程序之前，不妨运行一下程序，看看程序能够输出哪些信息。下面是这个程序执行一次的输出结果。

```
Hello from serial.
Thread number = 0
Hello from parallel. Thread number=0
Hello from parallel. Thread number=1
Hello from parallel. Thread number=3
```

```
Hello from parallel. Thread number=2
Hello from serial again.
```

源程序非常简单，具有 C 语言初步知识的程序员能够非常容易地读懂程序。结合源程序以及输出的结果，我们可以看到，这个程序首先从串行程序开始执行，然后开始分成四个线程开始并行执行。#pragma omp parallel 标志着一个并行区域的开始，在支持 OpenMP 的编译器中，根据线程的数目，随后的程序块会被编译到不同的线程中执行。而在不支持 OpenMP 的编译器中，这条编译制导语句会被简单忽略。因此，可以很方便地从串行程序切换到并行程序，不会给编程带来太多的负担。omp_get_thread_num()是一个 OpenMP 的库函数，用来获得当前线程号码。在 OpenMP 程序中，每一个线程会被分配给一个唯一的线程号码，用来标识不同的线程。在四个线程执行完毕之后，程序会重新退回到串行部分，打印最后一个语句，最后程序退出。如果我们设置 OMP_NUM_THREADS=1，则并行部分只会打印出一条信息。由于是多线程并行执行的，因此，多次执行可能产生不同的结果，下图就是三次执行的结果。

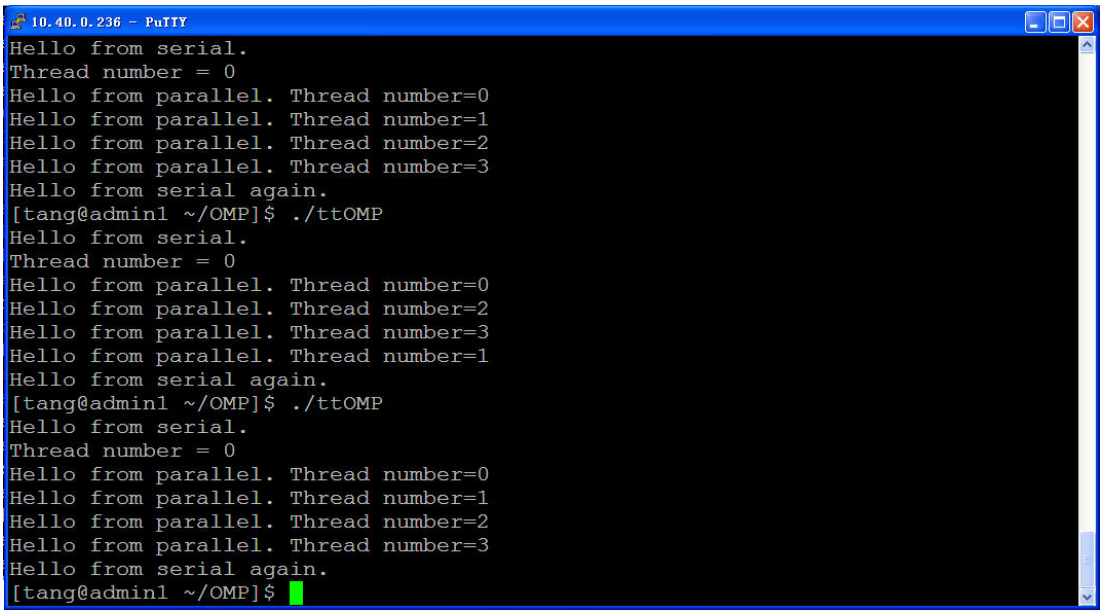

图 5.5 OpenMP 并行应用程序在多次执行过程中产生不同结果

以下各小节会对编程技术展开具体讨论。

5.2 OpenMP 多线程应用程序编程技术

5.2.1 循环并行化

循环并行化编译制导语句的格式：
循环并行化是使用 OpenMP 来并行化程序的最重要的部分，它是并行区域编程的一个特例。由于大量的科学计算程序将很大一部分时间用在处理循环计算上，对于循环进行并行化处理对这一部分应用程序非常关键，因此循环并行化是 OpenMP 应用程序中是一个相对独立且非

常重要的组成部分。在C/C++语言中，循环并行化语句的编译制导语句格式如下：

代码5.2 循环并行化语句的编译制导语句格式

```
#pragma omp parallel for [clause[clause…]]
    for (index = first ; test_expression ; increment_expr){
        body of the loop;
    }
或者
#pragma omp parallel [clause[clause]]
{
#pragma omp for [clause[clause]]
for (index = first; test_expression; increment_expr){
            body of the loop;
}
    }
```

这两个版本的效果基本相同。只是，如果并行的线程需要在循环的开始、或结束时作些工作的话，就只能用parallel与for子句分离的版本。

这个编译制导语句中的parallel关键字将紧跟的程序块扩展为若干完全等同的并行区域，每个线程拥有完全相同的并行区域；而关键字for则将循环中的工作分配到线程组中，线程组中的每一个线程完成循环中的一部分内容。编译制导语句的功能区域一直延伸到 for 循环、或用大括号{}包围起来的程序块的结束。编译制导语句后面的字句（clause）用来控制编译制导语句的具体行为，在后面将通过例子来详细讲解子句的构成以及相关子句的语法状况。

循环并行化语句的限制

并不是所有的循环语句都能够在其前面加上#pragma omp parallel来实现并行化，在OpenMP 2.5规范中，OpenMP对于可以以多线程执行的循环有以下约束：

•循环语句中的循环变量必须是有符号整型，如果是无符号整型，就无法使用。注意，在未来的OpenMP 3.0规范中，这个约束可能被取消。

• 循环语句中的比较操作必须是这样的形式：loop_variable <, <=, >或>= loop_invariant_integer。

•循环语句中的第三个表达式（for循环的循环步长）必须是整数加或整数减，加减的数值必须是一个循环不变量（loop invariant value）

•如果比较操作是<或<=，那么循环变量的值在每次迭代时都必须增加；相反地，如果比较操作是>或>=，那么循环变量的值在每次迭代时都必须减少。

•循环必须是单入口、单出口的，也就是说循环内部不允许有能够到达循环之外的跳转语句，也不允许有外部的跳转语句到达循环内部。在这里，exit 语句是一个特例，因为它将中止

整个程序的执行。如果使用了 goto 或 break 语句，那么它们的跳转范围必须在循环之内，不能跳出循环。异常处理也是如此，所有的异常都必须在循环内部处理。

虽然这些约束看上去有些多，但大多数循环都能够非常容易地被重写成符合约束条件地形式。只有满足上述约束条件，编译器才能通过 OpenMP 将循环并行化。然而，虽然编译器能够完成循环的并行化，仍然需要程序员来保证循环功能的正确。

简单循环并行化

我们首先看一个简单循环的并行化过程，将两个向量相加，并将计算的结果保存到第三个向量中，向量的维数为 n。向量相加即向量的各个分量分别相加。

```
for（int i=0;i<n;i++）
    z[i]=x[i]+y[i];
```

显然向量相加的算法中，各个分量之间没有数据相关性，循环计算的过程也没有循环依赖性，即某一次循环的结果不依赖于其它次循环的结果。而如下的例子当中就存在循环依赖性。

```
for（int i=0;i<n;i++）
    z[i]=z[i-1]+x[i]+y[i]
```

可以看出，第 i 次的循环依赖于 i-1 次的循环的结果，对于这样有依赖性的循环进行并行化必须考虑循环依赖性。

关于数据相关的概念，如果语句 S2 与语句 S1 存在数据相关，那么必然存在以下两种情况之一：

• S1 在循环的一次迭代中访问存储单元 L，而 S2 在随后的一次迭代中访问同一存储单元。

• S1 和 S2 在同一循环迭代中访问同一存储单元 L，但 S1 的执行在 S2 之前。

第一种情况称为循环迭代相关（loop-carried dependence），相关的存在是因为循环有多次迭代。而第二种情况是一种非循环迭代相关（loop-independent dependence），因为该相关的存在是由循环内代码的顺序决定的。

下面来看这样一个示例：d=1，n=99，迭代 k 中 S1 对存储单元 x[k]进行写操作，而迭代 k+1 中 S2 将读取该存储单元，这样就产生了一个循环迭代流相关。此外，迭代 k 中 S1 对存储单元 y[k-1]进行读操作，而迭代 k+1 中 S2 对其进行写操作，因此存在循环迭代反相关。在这种情况下，如果插入一条 parallel for 编译制导令该循环以多线程执行，就将得到错误的结果。

```
// 这段代码是错误的。由于存在循环迭代相关，该代码无法正常执行

x[0] = 0;
y[0] = 1;
```

```
#pragma omp parallel for private(k)
for (k = 1; k < 100; k++){
        x[k] = y[k-1] + 1; //S1
        y[k] = x[k-1] + 2; //S2
}
```

因为 OpenMP 编译制导是对编译器发出的命令，所以编译器会将该循环编译成多线程代码。但由于循环迭代相关的存在，多线程代码将不能成功执行。要解决此类问题，唯一的方法就是重写该循环，或选用另一个不包含循环迭代相关的算法。对于这个示例，可以预先确定 x[49]和 y[49]的初值，然后运用循环分块技术创建无循环迭代相关的循环 m。最后，插入 parallel for 并行化循环 m。通过这样的转换，原来的循环就可以在一个双核处理器系统上使用两个线程来执行了。

代码 5.3 使用分块技术将循环转换成等效的多线程代码

```
x[0] = 0;
y[0] = 1;
x[49] = 74; //根据等式 x(k) = x(k-2) + 3 计算得出
Computed according to equation x(k) = x(k-2) + 3
y[49] = 74 ; //根据等式 y(k) = y(k-2) + 3 计算得出
Computed according to equation y(k) =y(k-2) + 3

#pragma omp parallel for private(m, k)
for (m = 0; m < 2; m++){
        for (k = m*49 + 1; k < m*50 + 50; k++){
            x[k] = y[k-1] + 1; //S1
            y[k] = x[k-1] + 2; //S2
}
}
```

对于这个示例，除了使用 parallel for 编译制导，还可以使用 parallel sections 编译制导来并行化原本具有循环迭代相关的循环，以便能够在一个双核处理器系统上运行。

代码 5.4 使用 parallel sections 编译制导将循环转换成等效的多线程代码

```
#pragma omp parallel sections private(k)
{
#pragma omp section
        {
            x[0] = 0; y[0] = 1;
            for (k = 1; k < 49; k++){
                x[k] = y[k-1] + 1; //S1
                y[k] = x[k-1] + 2; //S2
}
}
#pragma omp section
```

```
{
            x[49] = 74; y[49] = 74;
            for (k = 50; k < 100; k++){
                x[k] = y[k-1] + 1; // S3
                y[k] = x[k-1] + 2; // S4
}
}
}
```

通过这个简单的示例，我们对一个存在循环迭代相关的循环进行了并行化，从中学习了一些行之有效的方法。有时候，在编写多线程代码时，为了充分利用双核或多核处理器，除了添加 OpenMP 编译制导以外，还需要进行简单的代码重构或变换。

循环并行化编译制导语句的子句

循环并行化子句可以包含一个或者多个子句来控制循环并行化的实际执行。有多个类型的子句可以用来控制循环并行化编译。最主要的子句是数据作用域子句。由于有多线程同时执行循环语句中的功能指令，这就涉及到数据的作用域问题。注意这里所说的作用域和 C/C++的数据作用域是不同的。这里的作用域用来控制某一个变量是否是在各个线程之间共享或者是某一个线程是私有的。数据的作用域子句用 shared 来表示一个变量是各个线程之间共享的，而用 private 来表示一个变量是每一个线程私有的，用 threadprivate 表示一个线程私有的全局变量。在 OpenMP 中，如果没有指定变量的作用域，则默认的变量作用域是共享的，这与 OpenMP 对应的共享内存空间编程模型是相互符合的。除了变量的作用域子句外，还有一些编译制导子句是用来控制线程的调度（schedule 子句），动态控制是否并行化（if 子句），进行同步的子句（ordered 子句）以及控制变量在串行部分与并行部分传递的子句（copyin 子句）。这些子句将在后面逐步介绍。

循环嵌套

在一个循环体内经常会包含另外一个循环体，循环产生了嵌套。在 OpenMP 中，我们可以将嵌套循环的任意一个循环体进行并行化。循环并行化编译制导语句可以加在任意一个循环之前，则对应的最近的循环语句被并行化，其它部分保持不变。因此，实际上并行化是作用于嵌套循环中的某一个循环，其它部分由执行到的线程负责执行。

代码 5.5 循环嵌套，并行化作用于外层循环

```
int i;int j
#pragma omp parallel for private (j)
for (i=0;i<2;i++)
       for (j=6;j<10;j++)
        printf ( “i=%d j=%d\n”,i,j) ;
```

执行结果：

```
i=0 j=6
i=1 j=6
i=0 j=7
```

```
i=1 j=7
i=0 j=8
i=1 j=8
i=1 j=9
i=0 j=9
```

代码 5.6 循环嵌套，并行化作用于内层循环

```
int i;int j;
for (i=0;i<2;i++)
#pragma omp parallel for
    for (j=6;j<10;j++)
        printf ("i=%d j=%d\n",i,j) ;
```

执行结果：：

```
i=0 j=6
i=0 j=8
i=0 j=9
i=0 j=7
i=1 j=6
i=1 j=8
i=1 j=7
i=1 j=9
```

在上述的程序中，程序 5.5 并行化作用于外层循环，程序 5.6 并行化作用于内层循环。在执行的过程中，并行执行的效果只与有作用的循环相关，在每一个并行执行线程的内部，程序继续按照串行执行。

控制数据的共享属性

OpenMP 程序在同一个共享内存空间上执行，由于可以任意使用这个共享内存空间上的变量进行线程间的数据传递，使得线程通信非常容易。一个线程可以写入一个变量而另外一个线程则可以读取这个变量来完成线程间的通信。

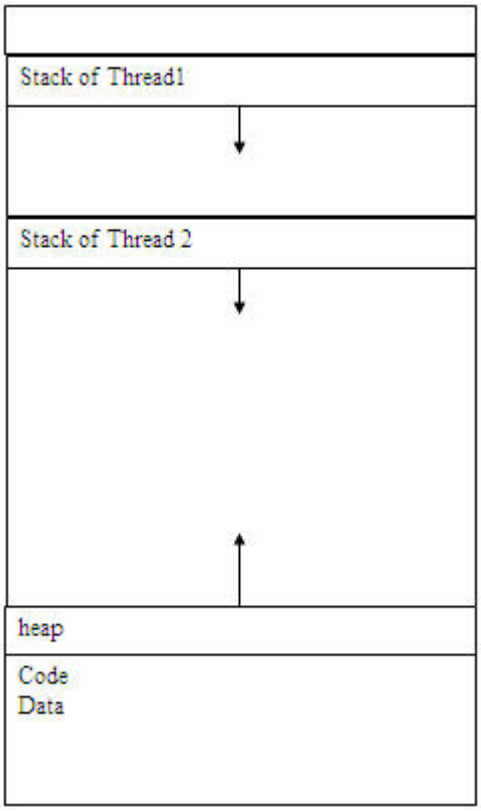

图 5. 6 多线程应用程序的内存分布结构

上述图 5.6 给出了多线程应用程序的内存分布结构。可以看出，每一个线程的栈空间都是私有的，因此分配在栈上的数据都是线程私有的，例如函数调用时分配的自动变量等。全局变量以及程序代码都是全局共享的；而动态分配的堆空间也是共享的。另外，在 OpenMP 中通过 threadprivate 来明确指出的某一个数据结构属于线程范围的全局变量。除了共享变量之外，OpenMP 还允许线程保留自己的私有变量不能让其它线程访问到。每一个线程会建立变量的私有拷贝，虽然变量名是相同的，实际上在共享内存空间内部的位置是不同的。因此，控制变量的作用域就非常重要。为了清楚地讨论两种不同的作用域，将 C/C++语言层面的作用域（即变量的是否 static, global, file level 等）称为“语法作用域”。数据作用域子句用来确定数据的共享属性，有下面四个子句。

shared 用来指示一个变量的作用域是共享的。private 用来指示一个变量作用域是私有的。firstprivate 和 lastprivate 分别对私有的变量进行初始化的操作和最后终结的操作。firstprivate 将串行的变量值拷贝到同名的私有变量中，在每一个线程开始执行的时候初始化一次。而 lastprivate 则将并行执行中的最后一次循环的私有变量值拷贝的同名的串行变量中。default 语句用来改变变量的默认私有属性。

在使用作用域子句的时候要遵循如下的一些规则。1）作用域子句作用的变量是已经申明的有名变量；2）作用域子句在作用到类或者结构的时候，必须作用到类或者结构的整体，而不能只作用于类或者结构的一个部分；3）一个编译制导语句能够包含多个数据作用域子句，但是变量只能出现在一个作用域子句中，即变量不能既是共享的，又是私有的。4）在语法结构上，作用域子句只能作用在出现在编译制导语句起作用的语句变量部分。另外，可以将作用域子句作用在类的静态变量上，例如 #pragma omp parallel for shared

（ClassName::_staticVariable）。由于整个类别可能被标记为私有的，则对类的构造有一定的特殊要求。例如，私有变量在分配空间的时候会被默认调用构造函数，则相应的类要求有默认的构造来初始化类变量；类变量在串行部分和并行部分需要进行数据传递的时候，则需要类提供相应的拷贝构造函数等。

除了上述显式申明变量作用域外，OpenMP 对不同的语法作用域规定了相应的默认数据作用域。默认情况下，并行区中所有的变量都是共享的，但有三种例外情况。首先是在 parallel for 循环中，循环索引变量是私有的。其次，那些并行区中的局部变量是私有的。第三，所有在 private，firstprivate，lastprivate 或 reduction 子句中列出的变量都是私有的。私有化是通过为每个线程创建各个变量的独立副本来完成的。

代码 5.7 数据的共享

```
int gval=8;
void funcb（int * x, int *y, int z）
{
        static int sv;
        int u;
        u=（*y）*gval;
        *x=u+z;
}

void funca（int * a, int n）
{
        int i;
        int cc=9;
        #pragma omp parallel for
        for（i=0;i<n;i++）{
            int temp=cc;
            funcb（&a[i],&temp,i）;
        }
}
```

在上述的例子中，函数 funca 调用了函数 funcb，并且在函数 funca 中使用了 OpenMP 进行并行化。全局变量 gval 是共享的。在 funca 函数的内部，变量 i 由于是循环控制变量，因此是线程私有的；cc 在并行化语句之外声明，是共享的；temp 在循环并行化语句内部的自动变量，是线程私有的；输入的指针变量 a 以及 n 是共享的，都在循环并行化语句之外声明。在函数 funcb 内部，静态变量 sv 是共享的，在程序内存空间中只有一份，因此，在这种使用方式下会引起数据冲突；变量 u 是自动变量，由于被并行线程调用，是线程私有的；参数 x 的本身是私有的指针变量，但是*x 指向的内存空间是共享的，其实际参数即函数 funca 中的 a 数组；参数 y 的本身是私有的指针变量，指向的*y 也是私有的，其实际内存空间即私有的 temp 占用的空间；数值参数 z 是线程私有的。

为了编程序方便，OpenMP 也提供了改变默认变量作用域的方法，即通过 default（shared）

将变量作用域变成默认共享的，或者使用 default（none）将默认作用域去掉，则需要显式的指定变量的作用域，否则会被认为错误。在 C/C++里面没有 default（shared）的作用域子句（在 Fortran 里面有），这是由于很多库函数为了效率使用了宏，对于 OpenMP 不能提供很好的支持，会造成程序的错误。

规约操作的并行化

一类经常需要并行化的计算是规约操作。在规约操作中，会反复将一个二元运算符应用在一个变量和另外一个值上，并把结果保存在原变量中。一个常见的规约操作就是数组求和，使用一个变量保存部分和，并把数组中的每一个值加到这个变量中，就可以得出最后所有数组的总和。OpenMP 对于这一类规约操作提供了特殊的支持，在使用规约操作时，只需在变量前指明规约操作的类型以及规约的变量。下面是一个规约操作的实例，分别计算两个数组中数值的总和。

代码 5.8 规约操作的并行化

```
# pragma omp parallel for private (arx,ary,n)  reduction (+:a,b)
for (i=0;i<n;i++) {
        a=a+arx[i];
        b=b+ary[i];
}
```

并不是所有的操作都能够使用规约操作。下面的表格列出了所有能够在 OpenMP 的 C/C++语言中出现的规约操作。

运算符	数据类型	默认初始值
+	Integer, floating point	0
*	Integer, floating point	1
-	Integer, floating point	0
&	Integer	所有位都开启，~0
|	Integer	0
^	Integer	0
&&	Integer	1
||	Integer	0

表 5.1 OpenMP 规约操作符及规约变量的初值

可以看到，规约操作只对语法内建的数值数据类型有效，对其他类型则无效。如果对其它类型或者用户自定义的类型，则必须使用同步语句来对共享变量进行保护。

对于在 reduction 子句中所指定的每一个变量，都会为每个线程创建一个私有副本，就象使用了 private 子句一样。此后私有副本的初值被设置为表 5.1 所列出的操作初值。在指定了 reduction 子句的区域或循环后，规约变量的原始值与各个私有副本的最终值进行指定的操作，然后用操作结果更新该规约变量的值。在尝试使用 reduction 子句进行多线程程序设计时，要记住以下三个要点：

•在第一个线程到达指定了 reduction 子句的共享区域或循环末尾时，原来的规约变量的值变为不确定，并保持此不确定状态直至规约计算完成。

• 如果在一个循环中使用到了 reduction 子句，同时又使用了 nowait 子句，那么在确保所有线程完成规约计算的栅栏同步操作前，原来的规约变量的值将一直保持不确定的状态。

•各个线程的私有副本值被规约的顺序是未指定的。因此，对于同一段程序的一次串行执行和一次并行执行，甚至两次并行执行来说，都无法保证得到完全相同的结果（这主要针对浮点计算而言），也无法保证计算过程中诸如浮点计算异常这样的行为会完全相同。

私有变量的初始化和终结操作

对于线程私有的变量，在每一个线程开始创建执行的时候其值是未确定的。当然，也可能通过 C++的构造函数来初始化类对象，但一般的内建类型的初始值都未定。对于把私有变量作为临时变量，并在线程内部的使用过程中初始化当然没有问题。然而某些情况下面，我们可能需要在循环并行化开始的时候访问到私有变量在主线程中的同名变量的值，也有可能需要将循环并行化最后一次循环的变量结果返回给主线程中的同名的变量，就好像我们正常地通过串行执行的效果一样。OpenMP 编译制导语句也对于这种需求给予支持，即使用 firstprivate 和 lastprivate 对这两种需求进行支持，使得循环并行开始执行的时候私有变量通过主线程中的变量初始化，同时循环并行结束的时候，将最后一次循环的相应变量赋值给主线程的变量。

代码 5.9 私有变量的初始化和终结操作

```
int val=8;
#pragma omp parallel for firstprivate（val） lastprivate（val）
for（int i=0;i<2;i++）{
    printf（"i=%d val=%d\n",i,val）;
    if（i==1）
        val=10000;
    printf（"i=%d val=%d\n",i,val）;
}
printf（"val=%d\n",val）;
```

下面是程序的执行结果

```
i=0 val=8
i=1 val=8
i=0 val=8
i=1 val=10000
val=10000
```

可以看到，在每一个线程的内部，私有变量 val 被初始化为主线程原有的同名变量的值，并且在循环并行化退出的时候，相应的变量被原有串行执行的最后一次执行对应的值所赋值。

数据相关性与并行化操作

并不是所有的循环都能够使用#pragma omp parallel for 来进行并行化。为了对一个循环进行并行化操作，我们必须要保证数据两次循环之间不存在数据相关性。数据相关性又被称为数据竞争（Data Race）。当两个线程对同一个变量进行操作，并且有一个操作为写操作的时候，就说明这两个线程存在数据竞争。此时，读出的数据不一定就是前一次写操作的数据，而写入的数据也可能并不是程序所需要的。下面是一个典型的具有数据相关性的循环，不能通过直接采用#pragma omp parallel for 的方式将其并行化，因为两个相邻的循环会使用到相同的内存空间中的数据，并且有一个操作为写操作，无法保持数据的一致性。

```
for（int i=0;i<99;i++）
    a[i]=a[i]+a[i+1];
```

为了将一个循环并行化，而不影响程序的正确性，需要仔细检查程序使得程序在并行化之后，两个线程之间不能够出现数据竞争。当然，有时候根据应用的特殊需求，在能够保证得出正确结果的情况下，可以允许存在数据竞争，并将循环并行化。在出现数据竞争的时候，我们可以通过增加适当的同步操作，或者通过程序改写来消除竞争。

下面是一个具有循环之间数据相关性的串行程序

```
for（int j=1;j<N;j++）
    for（int i=0;i<N;i++）
        a[i,j]=a[i,j]+a[i,j-1];
```

但是，由于内部循环的过程中，j 的值是固定的，对于内部循环来说，循环之间没有数据的相关性，可以将内部循环进行并行化，得到我们的并行化版本。

代码 5.10 具有循环之间数据相关性的并行程序

```
for（int j=1;j<N;j++）
#pragma omp parallel for
    for（int i=0;i<N;i++）
        a[i,j]=a[i,j]+a[i,j-1];
```

可以看到，在进行并行化的过程中，我们必须要仔细分析循环之间的数据相关性，在某些时候，可以通过程序改写，消除产生数据相关性的原因，才能获得并行应用程序。

5.2.2 并行区域编程

上面我们曾经说过循环并行化实际上是并行区域编程的一个特例，在这一小节我们将详细讨论非循环的并行区域编程。并行区域简单的说就是通过循环并行化编译制导语句使得一段代码能够在多个线程内部同时执行。本章的第一个例子就是一个并行区域的例子，根据环境变量的设置，该程序在相应的线程内部打印出信息。

并行区域编译制导语句的格式与使用限制

在 C/C++语言中，并行区域编写的格式如下所示：

```
#pragma omp parallel [clause[clause]…]
    block
```

其中 block 是需要在多个线程中执行的代码块，每一个线程在遇到并行区域的编译制导语句的时候，都会同时执行跟随其后的程序代码块。在并行区域的编译制导语句后面也可以跟随一些子句，包括 private，shared，default，reduction，if，copyin 等子句都可以在并行区域编译制导语句中出现。在后面的叙述中，我们将介绍每一个子句的功能和作用。parallel 编译制导语句与循环并行化 parallel for 语句类似，在使用到程序块之前也有一定的限制。程序块必须是一个只有单一入口和单一出口的程序块，不能从外面转入到程序块的内部，也不允许从程序块内部有多个出口转到程序块之外，在程序块内部的跳转是允许的。在程序块内部直接调用 exit 函数来退出整个程序的执行也是允许的。

parallel 编译制导语句的执行过程
为了理解编译制导语句 parallel 语句的执行过程，我们先看看如下两个程序的执行结果。这两个程序非常简单，且相似，唯一的不同之处在于前面一个程序的编译制导语句是并行区域编译制导语句 parallel 而后面一个程序的编译制导语句是循环并行化的编译制导语句 parallel for。

代码 5.11parallel 编译制导语句 1

```
#pragma omp parallel
    for（int i=0;i<5;i++）
        printf（"hello world i=%d\n",i）;
```

程序的执行结果：

```
hello world i=0
hello world i=0
hello world i=1
hello world i=1
hello world i=2
hello world i=2
hello world i=3
hello world i=3
hello world i=4
hello world i=4
```

代码 5.12parallel 编译制导语句 2

```
#pragma omp parallel for
    for（int i=0;i<5;i++）
        printf（"hello world i=%d\n",i）;
```

程序的执行结果：

```
hello world i=0
hello world i=3
hello world i=1
hello world i=4
hello world i=2
```

可以看到，两个程序唯一的区别在于程序中黑体标出的 for。(在执行的过程中，环境变量OMP_NUM_THREADS=2。)从这两个执行结果中我们可以明显地看到并行区域与循环并行化的区别，即并行区域采用了复制执行的方式，将代码在所有的线程内部都执行一次；而循环并行化则采用了工作分配的执行方式,将循环所需要的所有工作量按照一定的方式分配到各个执行线程中，所有线程执行工作的总和是原先串行执行所完成的工作量。

总结上述的并行区域 parallel 语句的作用是当程序遇到 parallel 编译制导语句的时候,就会生成相应数目（根据环境变量）的线程组成一个线程组，并将代码重复地在各个线程内部执行。parallel 的末尾有一个隐含的同步屏障（barrier），所有线程完成所需的重复任务有，在这个同步屏障出会和（join)。此时，线程组的主线程（master）继续执行，而相应的子线程（slave）则停止执行。

线程私有数据与 threadprivate，copyin 子句

前面我们已经看到在循环并行化的过程中,有一套默认的规则用来控制变量在线程之间的共享属性。除了 private 子句能够产生线程私有的变量之外，还需要考虑一些全局的数据。这些全局的数据可能是在整个程序运行过程中都需要的数据,或者是在源程序中跨多个文件所需要的变量。在通常的情况下，这些数据都是共享的数据，所有的线程访问的都是共享内存空间中的同一内存地址内容。然而，有的时候，对于每一个线程来说，可能需要生成自己私有的线程数据,此时,就需要使用 threadprivate 子句用来标明某一个变量是线程私有数据，在程序运行的过程中，不能够被其它线程访问到。

代码 5.13 线程私有数据与 threadprivate 子句

```
int counter=0;
#pragma omp threadprivate (counter)     //using threadprivate

void inc_counter ()
{
    counter++;
}

int main (int argc, char * argv[])
{
    #pragma omp parallel
    for (int i=0;i<10000;i++)
        inc_counter () ;

    printf ("counter=%d\n",counter) ;
```

```
}
```

在上述的程序例子中，我们使用了 threadprivate 语句（使用黑体并有注释的语句），使得 counter 变成了每一个线程拥有的私有变量。此时的执行过程最后结果都为：

```
counter=10000
```

而如果将含有注释的那一行删除，就将全局变量 counter 变为共享，执行过程中会产生数据冲突，执行结果不可预知，下面是可能的一次执行结果：

```
counter=15194
```

可以明显看到出现了数据相关性。

对于所有的线程私有全局变量来说，除了主线程，其它线程的私有变量在运行过程中是没有初始化的，为了使用主线程的变量初始化的值，我们使用 copyin 子句对线程私有的全局变量进行初始化。

代码 5.14 线程私有数据与 copyin 子句

```
int global=0;
#pragma omp threadprivate (global)
int main (int argc, char * argv[])
{
    global=1000;
    #pragma omp parallel copyin (global)
    {
        printf ("global=%d\n",global) ;
        global=omp_get_thread_num () ;
    }
    printf ("global=%d\n",global) ;
    printf ("parallel again\n") ;
    #pragma omp parallel
        printf ("global=%d\n",global) ;
}
```

程序的执行结果为：

```
global=1000
global=1000
global=0
parallel again
global=0
```

global=1

可以看出，通过 copyin 的操作，确实将线程的私有变量初始化为主线程中相应的全局变量的值。在并行区域执行完毕退出后，主线程与子线程中的相应的全局变量继续有效，并且在在一次进入并行区域时，使用上一次退出时所赋的值。

并行区域之间的工作共享

上述的 parallel 编译制导语句提供了在 OpenMP 程序中一种简单的并行方法，能够将工作在多个线程之间重复运行。实际上，对于并行程序来说，相对于重复地做同样一件工作，更有意义的是将一定的工作量分配到各个线程，使得各个线程共同完成更大的工作量。因此，并行程序希望能够在不同的数据集上执行相同的计算，甚至希望并行执行完全不同的工作。在通常的并行程序编写中，一般会使用工作队列的方式将工作放置到一个队列中，每一个线程每次从队列中获取一件工作。而在 OpenMP 程序中，每一个线程都可以调用 omp_get_thread_num（）来得到自己唯一的线程号，因此可以利用这个线程号来获得不同的工作任务执行。实际上，在 OpenMP 支持的语法中，可以直接使用编译制导语句 for 将任务分配到各个线程，就像前一节所说的循环并行化一样；另外，也可以用 sections 编译制导语句以及 section 子句自然地将不同的工作任务编写成不同的代码片段并行执行。在下面的内容中，我们针对每一种编程方式给出例子讲解如何进行工作共享的并行区域编程。

工作队列

工作队列的基本工作过程即为维持一个工作的队列，线程在并行执行的时候，不断从这个队列中取出相应的工作完成，直到队列为空为止。

代码 5.15 工作队列

```
int next_task=0;
int get_next_task ()
{
        int task;
        #pragma omp critical //用来做同步操作 Used to conduct synchronization
        if (next_task<MAX_TASK) {
            task=next_task;
            next_task++;
        }else
            task=-1;
        return task;
}

void task_queue ()
{
        int my_task;
        #pragma omp parallel private (my_task)
        {
            my_task=get_next_task () ;
            while (my_task!=-1) {
```

```
                get_task_done (my_task) ;
                my_task=get_next_task () ;
            }
        }
}
```

上述程序的并行部分不断从一个任务队列中取出相应的任务完成，直到完成任务队列中的所有任务。

根据线程号分配任务

由于每一个线程在执行的过程中的线程标识号是不同的，可以根据这个线程标识号来分配不同的任务，下面的例子程序就演示了如何根据线程标识号来完成不同的任务。

代码 5.16 根据线程号分配任务

```
#pragma omp parallel private (myid)
{
nthreads=omp_get_num_threads () ;
myid=omp_get_thread_num () ;
get_my_work_done (myid,nthreads) ;
}
```

在上述的程序中，首先获得当前所有线程的数目，并且根据线程的总数以及相应的线程标识号来确定相应的工作，完成任务的并行分配。

使用循环语句分配任务

由于通过循环并行化在线程之间分配相应的任务已经在上一个小节详细讲述，在这里就不进一步举例子说明了。实际上，循环并行化是可以单独在并行化区域中出现的，每一个循环中的任务就被分配到各个工作线程中。

代码 5.17 使用循环语句分配任务

```
#pragma omp parallel
    {
        printf ("outside loop thread=%d\n",omp_get_thread_num ()) ;
#pragma omp for
        for (int i=0;i<4;i++)
            printf ("inside loop i=%d thread=%d\n",i,omp_get_thread_num ()) ;
    }
```

程序的运行结果：

```
outside loop thread=0
outside loop thread=1
inside loop i=2 thread=1
```

```
inside loop i=0 thread=0
inside loop i=3 thread=1
inside loop i=1 thread=0
```

可以看出，在循环的外部，程序代码被各个线程复制执行，而在循环的内部，循环的所有任务被各个线程分别完成。从总体上来说，循环执行的次数与串行执行的次数一致。

上述的工作空间的共享实际上是编程人员通过自己的经验，使用 OpenMP 一些接口编写自己所需的任务分配代码。实际上，在 OpenMP 编程规范中已经对能够在不同的线程中执行不同的任务有所支持。使用工作分区（sections）的方法就能够达到这一点。

工作分区编码（sections）

下面是一个工作分区编码的示例

代码 5.18 一个工作分区编码的示例

```
#pragma omp parallel sections
    {
#pragma omp section
        printf ("section 1 thread=%d\n",omp_get_thread_num ()) ;
#pragma omp section
        printf ("section 2 thread=%d\n",omp_get_thread_num ()) ;
#pragma omp section
        printf ("sectino 3 thread=%d\n",omp_get_thread_num ()) ;
    }
```

程序运行结果为：

```
section 1 thread=0
section 2 thread=1
sectino 3 thread=0
```

可以看到，在使用工作分区编码的时候，各个线程自动从各个分区中获得任务执行。并且在执行完一个分区的时候，如果分区组里面还有未完成的工作，则继续取得任务完成。

5.2.3 线程同步

在 OpenMP 应用程序中，由于是多线程执行，必须要有必要的线程同步机制以保证程序在出现数据竞争的时候能够得出正确的结果，并且在适当的时候控制线程的执行顺序，以保证执行结果的确定性。OpenMP 支持两种不同类型的线程同步机制，一种是互斥锁的机制，可以用来保护一块共享的存储空间，使得每一次访问这块共享内存空间的线程最多一个，保证了数据的完整性；另外一种同步机制是事件通知机制，这种机制保证了多个线程之间的执行顺序。互斥的操作针对需要保护的数据而言，在产生了数据竞争的内存区域加入互斥，包括

critical，atomic 等语句以及由函数库中的互斥函数构成的标准例程。而事件机制则在控制规定线程执行顺序时所需要的同步屏障（barrier），定序区段（ordered sections），主线程执行（master）等。OpenMP 也对用户自主构成的同步操作提供了一定的支持。

数据竞争

我们看一个简单的数据竞争的例子，即通过下述的算法来寻找一个正整数数组中最大的元素。串行的算法如下所示，假设数组为 ar，数组元素的数目为 n。

```
int i; int max_num=-1;
for (i=0;i<n;i++)
    if (ar[i]>max_num)
        max_num=ar[i];
```

对于这样一段简单的代码，我们可以直接在 for 循环前面加入循环并行化的编译制导语句使得整个程序代码段可以并行执行。

```
int i; int max_num=-1;
#pragma omp parallel for
for (i=0;i<n;i++)
    if (ar[i]>max_num)
        max_num=ar[i];
```

但是由于是多线程执行，这样的并行执行代码是不正确的，有可能产生错误的结果。下面是一个可能的执行结果，假设数组有两个元素，分别为 2 和 3，系统中有两个线程在执行，则每个线程只需对一个元素进行判断即可。线程 1 在执行的过程中，发现 0 号元素比 max_num 要大，需要将 2 赋值给 max_num。恰在此时，系统将线程 1 挂起。线程 2 继续执行，发现 1 号元素也比 max_num 大，执行的结果就是 max_num=3。线程 2 执行完自己的任务后，会同步在一个隐含的屏障上（barrier）。线程 1 被唤醒，由于它已经过了整数判断的阶段，因此它直接将 0 号元素赋值给 max_num，使得 max_num=2。执行的结果与串行结果完全不同。产生结果出现错误的主要原因就是我们有超过两个线程同时访问一个内存区域，并且至少有一个线程的操作是写操作，这样就产生了数据竞争。如果不对数据竞争进行处理的话，就会产生执行结果出错。

互斥锁机制

在 OpenMP 中，提供了三种不同的互斥锁机制用来对一块内存进行保护，它们分别是临界区（critical），原子操作（atomic）以及由库函数来提供同步操作。

临界区

临界区通过编译制导语句对产生数据竞争的内存变量进行保护。在程序需要访问可能产生竞争的内存数据的时候，都需要插入相应的临界区代码。临界区编译制导语句的格式如下所示：

```
#pragma omp critical [（name）]
    block
```

如此，在执行上述的程序块 block 之前，必须首先要获得临界区的控制权。在线程组执行的时候，运行时库会保证每次最多只有一个线程执行临界区。name 是一个临界区的属性，给临界区一个命名。在对不同的内存区域进行保护的时候，如果两个线程实际上访问的并不是同一块内存区域，这是不会产生冲突的，没有必要对它们之间进行互斥锁的操作。因此，在 OpenMP 中就提供对临界区的命名操作。可以将不同命名的临界区保护不同的内存区域，如此就可以在访问不同内存区域的时候使用不同命名的临界区。运行时库是不会在不同命名的临界区之间进行互斥锁同步操作的。

因此，修改上述寻找正整数数组最大的元素的代码就可以修改成如下的形式。

代码 5.19 正整数数组最大的元素的临界区版本

```
int i; int
max_num_x=max_num_y=-1;
#pragma omp parallel for
for (i=0;i<n;i++)
{
#pragma omp critical  (max_arx)
    if (arx[i]>max_num_x)
        max_num_x=arx[i];

#pragma omp critical  (max_ary)
    if (ary[i]>max_num_y)
        max_num_y=ary[i];
}
```

在这里稍微修改了上述的代码，用来寻找两个数组各自的最大元素。通过两个不同的命名临界区 max_arx 与 max_ary 进行互斥操作因为两个数组之间并不存在数据相关性，也不存在数据竞争。但是上述的代码效率并不高，对于两个数组来说实际上分别进行了线性的操作，在循环内部进行的同步过多，在后续的内容中我们将提供如何提高效率的方法。

原子操作

原子操作是 OpenMP 编程方式给同步编程带来的特殊的编程功能，通过编译制导语句的方式直接获取了现在多处理器计算机体系结构的功能。我们知道，现代体系结构的多处理计算机提供了原子更新一个单一内存单元的方法，即通过单一一条指令就能够完成数据的读取与更新操作，例如在英特尔平台上的 CMPXCHG 的指令能够同时完成数据比较和交换功能。所有这些功能可以被成为原子操作，即操作在执行的过程中是不会被打断的。因此，通过这种方式就能够完成对单一内存单元的更新，提供了一种更高效率的互斥锁机制。在 OpenMP 的程序中，这样一种先进的功能被通过#pragma omp atomic 编译制导语句提供。值得注意的是，上面讲述的临界区操作能够作用在任意的代码块上，而原子操作只能作用在语言内建的基本数据结构。在 C/C++语言中，原子操作的语法格式如下所示

```
#pragma omp atomic
    x <binop>=expr
```

或者

```
#pragma omp atomic
    x++//or x--, --x, ++x
```

明显的，能够使用原子语句的前提条件是相应的语句块能够转化成一条机器指令，使得相应的功能能够一次执行完毕而不会被打断。下面是在 C/C++语言中可能的原子操作。

“+ * - / & ^ | << >>”

值得注意的是，当对一个数据进行原子操作保护的时候，就不能对数据进行临界区的保护，因为这是两种完全不同的保护机制，OpenMP 运行时并不能在这两种保护机制之间建立配合机制。因此，用户在针对同一个内存单元使用原子操作的时候需要在程序的所有涉及到的部位都加入原子操作的支持。

代码 5.20 原子操作

```
int counter=0;
#pragma omp parallel
{
for (int i=0;i<10000;i++)
     #pragma omp atomic  //atomic operation
         counter++;
}
printf ("counter = %d\n",counter) ;
```

注意上述程序中的黑体的语句，当黑体语句有效的时候，即使用 atomic 语句，则最后的执行结果都是一致的，执行结果总是为下面的形式（使用两个线程执行）：

counter = 20000

而当这一行语句被从源程序中删除时，由于有了数据的相关性，最后的执行结果是不确定的，下面是一个可能的执行结果：

counter = 12014

运行时库函数的互斥锁支持

除了上述的 critical 与 atomic 编译制导语句，OpenMP 还通过一系列的库函数支持更加细致的互斥锁操作，方便使用者满足特定的同步要求。下面的表格列出了由 OpenMP 库函数提供的互斥锁函数。

函数名称	描述
void omp_init_lock (omp_lock_t *)	初始化一个互斥锁
void omp_destroy_lock (omp_lock_t*)	结束一个互斥锁的使用并释放内存
void omp_set_lock (omp_lock_t *)	获得一个互斥锁

void omp_unset_lock (omp_lock_t *)	释放一个互斥锁
int omp_test_lock (omp_lock_t *)	试图获得一个互斥锁，并在成功时返回真（true），失败时返回假（false）

表 5. 1 OpenMP 库函数提供的互斥锁函数

上述库函数的使用比一般的编译制导语句更加灵活。编译制导语句进行的互斥锁支持只能放置在一段代码之前，作用在这段代码之上。使用运行库函数的互斥锁支持例程则可以将函数置于程序员所需的任意位置。程序员必须自己保证在调用相应锁操作之后释放相应的锁，否则就会造成多线程程序的死锁。另外，运行库函数还支持嵌套的锁机制。需要支持嵌套锁机制的原因是由于在某些情况下，例如进行递归函数调用的时候，同一个线程需要获得一个互斥锁多次。如果互斥锁不支持嵌套调用的话，在同一个互斥锁上调用两次获得锁的操作而中间没有释放锁的操作，将会造成线程死锁。因此，在 OpenMP 里支持同一个线程对锁的嵌套调用。嵌套调用同一个锁必须使用如下特殊的嵌套锁操作，嵌套锁操作的库函数与上述锁操作类似，不同的地方是在每一个函数都包含了 nest。

函数名称	描述
void omp_init_nest_lock (omp_lock_t *)	初始化一个嵌套互斥锁
void omp_destroy_nest_lock (omp_lock_t*)	结束一个嵌套互斥锁的使用并释放内存
void omp_set_nest_lock (omp_lock_t *)	获得一个嵌套互斥锁
void omp_unset_nest_lock (omp_lock_t *)	释放一个嵌套互斥锁
int omp_test_nest_lock (omp_lock_t *)	试图获得一个嵌套互斥锁，并在成功是返回真（true），失败是返回假（false）

表 5. 2 OpenMP 库函数提供的内嵌的互斥锁函数

下面一个简单的例子说明了如何使用锁机制来控制对一个计数器的访问：

代码 5.21 使用锁机制来控制对一个计数器的访问

```
omp_lock_t lock;
int counter=0;

void inc_counter ()
{
        printf ("thread id=%d\n",omp_get_thread_num ()) ;
        for (int i=0;i<1000;i++) {
            omp_set_nest_lock (&lock) ;
            counter++;
            omp_unset_nest_lock (&lock) ;
        }
}

void dec_counter ()
{
        printf ("thread id=%d\n",omp_get_thread_num ()) ;
```

```
        for (int i=0;i<1000;i++) {
            omp_set_nest_lock (&lock) ;
            counter--;
            omp_unset_nest_lock (&lock) ;
        }
}

int _tmain (int argc, _TCHAR * argv[])
{
        omp_init_nest_lock (&lock) ;
#pragma omp parallel sections
        {
#pragma omp section
            inc_counter () ;
#pragma omp section
            dec_counter () ;
        }
        omp_destroy_nest_lock (&lock) ;
        printf ("counter=%d\n",counter) ;
}
```

可以看到,这里有一个线程不断地增加计数器的值,而另外一个线程不断地减少计数器的值,因此需要同步的操作。这里演示如何使用嵌套型的锁函数,改成普通的锁函数的效果是一样的。但是在某些情况下,如果一个线程必须要同时加锁两次,则只能使用嵌套型的锁函数。

事件同步机制

事件同步机制与上述的锁机制不同,锁机制是为了维护一块代码或者一块内存的一致性,使得所有在其上的操作串行化;而事件同步机制则用来控制代码的执行顺序,使得某一部分代码必须在其它的代码执行完毕之后才能执行。

隐含的同步屏障(barrier)

在每一个并行区域都会有一个隐含的同步屏障(barrier),执行此并行区域的线程组在执行完毕本区域代码之前,都需要同步并行区域的所有线程。一个同步屏障要求所有的线程执行到此屏障,然后才能够继续执行下面的代码。#pragma omp for,#pragma omp single, #pragma omp sections 程序块都包含自己的隐含的同步屏障。为了避免在循环过程中不必要的同步屏障, 可以增加 nowait 子句到相应的编译制导语句中。例如在如下的代码中:

代码 5.22 隐含的同步屏障

```
        #pragma omp parallel
        {
        #pragma omp for nowait
        for (int i = 1; i < size; ++i)
```

```
        x[i] =  (y[i] + z[i]) /2;
                printf ( “finished\n” ) ;
        }
```

在工作共享的 for 循环结束后，并不需要等待其它线程的同步操作，而可以继续执行下面的打印操作。但是，在并行区域的结束还是会有一个隐含的同步屏障，这是所有的线程需要同步的地方。

明确的同步屏障语句

在并行执行的时候，在有些情况下，隐含的同步屏障并不能提供有效的同步措施，程序员可以在需要的地方插入明确的同步屏障语句#pragma omp barrier。此时，在并行区域的执行过程中，所有的执行线程都会在同步屏障语句上进行同步。

```
#pragma omp parallel
{
        initialization () ;

        #pragma omp barrier

        process () ;
}
```

上述例子中，只有等所有的线程都完成初始化操作以后，才能够进行下一步的处理动作，因此，在此处插入一个明确的同步屏障操作以完成线程之间的同步。下面的程序是对上述程序的扩展，是一个实际可运行的例子。

代码 5.23 明确的同步屏障语句

```
 void initialization ()
 {
         int counter=0;
         printf ("thread %d start initialization\n",omp_get_thread_num ()) ;
         for (int i=0;i<100000;i++)
             counter++;
         printf ("thread %d finish initialization\n",omp_get_thread_num ()) ;
 }

 void process ()
 {
         int counter=0;
         printf ("thread %d start process\n",omp_get_thread_num ()) ;
         for (int i=0;i<100000;i++)
             counter+=i;
         printf ("thread %d finish process\n",omp_get_thread_num ()) ;
```

```
}

int main (int argc, char * argv[])
{
#pragma omp parallel
        {
            initialization () ;
#pragma omp barrier
            process () ;
        }
}
```

为了演示实际的屏障效果，我们将运行的线程数目增加到 5 个（OMP_NUM_THREADS=5），下面是某一次执行的结果：

```
thread 2 start initialization
thread 1 start initialization
thread 2 finish initialization
thread 3 start initialization
thread 4 start initialization
thread 0 start initialization
thread 0 finish initialization
thread 3 finish initialization
thread 4 finish initialization
thread 1 finish initialization
thread 2 start process
thread 1 start process
thread 4 start process
thread 3 start process
thread 4 finish process
thread 2 finish process
thread 1 finish process
thread 0 start process
thread 3 finish process
thread 0 finish process
```

可以看到，在并行区域的内部，由于加入了明确的屏障语句，直到等到所有的线程执行初始化完毕之后，才真正进入处理的操作。

循环并行化中的顺序语句（ordered）

在某些情况下，我们对于循环并行化中的某些处理需要规定执行的顺序，典型的情况是在一次循环的过程中，一大部分的工作是可以并行执行的，而其余的工作需要等到前面的工作全

部完成之后才能够执行。在循环并行化的过程中，可以使用 ordered 子句使得顺序执行的语句直到前面的循环都执行完毕之后再执行。下面的例子说明了 ordered 子句是如何对结果产生影响的。

代码 5.24 循环并行化中的顺序语句

```
void work (int k)
{
        printf ("thread id =%d k=%d\n",omp_get_thread_num () ,k) ;
#pragma omp ordered
        printf (" %d\n", k) ;
}
void ordered_func (int lb, int ub, int stride)
{
        int i;
#pragma omp parallel for ordered schedule (dynamic)
        for  (i=lb; i<ub; i+=stride)
            work (i) ;
}
int main (int argc, char ** argv)
{
        ordered_func (0, 50, 5) ;
        return 0;
}
```

这个程序根据步长为 5 的等差数列进行工作，下面是一次执行的结果(OMP_NUM_THREADS=5)：

```
thread id =4 k=0
 0
thread id =4 k=25
thread id =0 k=15
thread id =2 k=20
thread id =1 k=5
 5
thread id =1 k=30
thread id =3 k=10
 10
thread id =3 k=35
 15
thread id =0 k=40
 20
thread id =2 k=45
 25
 30
 35
```

40
45

从结果我们可以看出，虽然在 ordered 子句之前的工作是并行执行的，但是在遇到 ordered 子句的时候，只有前面的循环都执行完毕之后，才能够进行下一步的执行。在使用事件进行执行顺序处理的同步操作时候，OpenMP 还提供了 master 子句（只能在主线程中执行）以及 flush 子句（用于程序员自己构造执行顺序）等，限于篇幅，就不再赘述。

循环调度与分块
在多线程程序中，要实现较好的负载平衡而获得最优的性能，就必须对循环进行高效的调度与分块。这样做的最终目的是保证执行核尽可能地在大部分时间内保持忙碌状态，同时将调度开销、上下文切换开销和同步开销降到最低。如果负载平衡做的很差，那么某些线程可能很早就完成了自己的工作，从而导致处理器资源闲置，损失了性能。为了提供一种简单的方法能够在多个处理器核之间调节工作负载，OpenMP 给出了四种调度方案，可以适用于很多情况：static、dynamic、runtime 和 guided。英特尔 C++和 Fortran 编译器都支持这四种调度方案。

负载平衡很差通常是由循环迭代计算时间的不确定性引起的。一般来说，通过检查源代码的方法来确定循环迭代计算时间并不是太难。在多数情况下，循环迭代总是耗费一定数量的时间。即便不是这样，也可以找到耗时相近的一组迭代。例如，有时候所有的偶数迭代集合和所有奇数迭代集合所耗费的时间几乎相等，或者循环前半部分迭代和后半部分迭代所耗费的时间几乎相等。另一方面，要找出耗时相同的迭代集合几乎是不可能的。然而不管怎样，都可以通过使用 schedule(kind [, chunksize])子句来提供循环调度信息，使编译器和运行时库能够更好的划分迭代并将迭代分布到各个线程（也就是处理器核），从而实现更好的负载平衡。

默认情况下，OpenMP 的 parallel for 循环或任务分配 for 循环都会采用静态负载平衡调度策略（static-even）。这就意味着该循环的迭代将以近乎平均的方式分布到各个线程上。如果有 m 次迭代，线程组中有 N 个线程，那么每个线程就执行 m/N 次迭代。编译器和运行时库将正确处理 m 不能整除 N 的情况。

使用静态平衡调度，能够尽可能地降低当多个处理器同时访问同一片内存区域时发生访存冲突的几率。这种方案之所以可行，是因为循环一般是顺序访存的，所以将循环分割为较大的块就可以减少重叠访存的几率，提高处理器 cache 的使用效率。考虑下面这个简单的循环使用静态平衡策略在两个线程上执行的情形。

```
#pragma omp parallel for
for (k = 0; k < 1000; k++) do_work(k);
```

OpenMP 将在一个线程上执行迭代 0 到 499，在另一个线程上执行迭代 500 到 999。虽然这种划分方式对于内存的利用来说可能是比较好的，但对于负载平衡可能是不利的。而更不幸的是，这句话反过来也成立：有利于负载平衡的策略也有可能对访存的性能不利。因此，进行性能优化时，就必须在优化内存利用和优化负载平衡之间进行折中，通过对性能的测量找出能够得到最佳结果的方法。

在 OpenMP 的 for 结构中，使用 schedule 子句将循环调度和分块信息传达给编译器和运行时库。

```
#pragma omp for schedule(kind [, chunksize])
```

表 5.4 总结了 OpenMP 规范中所指定的四种调度方案。如果指定了可选参数 chunksize（块大小），则该参数必须是不随循环变化的正常数常量或整数表达式。

在调整块大小时要特别注意，因为它可能会对性能带来负面影响。随着块的大小的减小，线程用于从任务队列中获得任务的时间会增加，结果使访问任务队列的开销增加，而降低性能，并有可能抵消负载平衡带来的性能提升。

调度类型	描述
static（默认不指定块大小）	将所有循环迭代划分成相等大小的块，或在循环迭代次数不能整除线程数与块大小的乘积时划分成尽可能大小相等的块。如果没有指定块大小，迭代的划分将尽可能的平均，使每个线程分到一块。将块大小设置成 1 就可以交错分配各个迭代
dynamic	使用一个内部任务队列，当某个线程可用时，为其分配由块大小所指定的一定数量的循环迭代。线程完成其当前分配的块后，将从任务队列头部取出下一组迭代。在默认情况下，块大小是 1。使用这种调度类型时要小心，因为这种调度策略需要额外的开销
guided	与 dynamic 策略类似，但块大小刚开始比较大，后来逐渐减小，从而减少了线程用于访问任务队列的时间。可选的 chunk 参数可以指定所使用的块大小的最小值，默认是 1
runtime	在运行时使用 OMP_SCHEDULE 环境变量来确定使用上述调度策略中的某一种。OMP_SCHEDULE 是一个字符串，其格式和 parallel 结构中所给出的调度策略参数格式相同

表 5.3 OpenMP 中的四种调度方案

对于 dynamic 调度来说，块是以先来先服务的方式进行处理的，默认的块大小是 1。每一次取得的迭代次数和 schedule 子句中所指定的块大小相等，但最后一个块例外。当一个线程执行完分配给它的迭代后，它将请求另一组迭代，其数量由块大小指定。这个过程不断重复，直至所有的迭代都被完成。最后一组迭代的个数可能小于块大小。例如，如果使用 schedule(dynamic, 16)子句将块大小定义为 16，而总的迭代次数是 100，那么划分的情况就是 16，16，16，16，16，16，4，共分成 7 个块。

对于 guided 策略来说，一个循环的划分是基于下列公式来完成的，其中初值 β_0 ＝迭代次数。

$$\pi_k = \left\lceil \frac{\beta_k}{2N} \right\rceil$$

其中 N 是线程个数，π_k 代表第 k 块的大小，从第 0 块开始，在计算第 k 块的大小时，β_k 代表剩下的未调度的循环迭代次数。

如果 π_k 的值太小，那么该值就会被块大小 S 所取代，而 S 是由 schedule(guided, chunk-size) 子句指定的。如果在 schedule 子句中没有指定块大小的值，则取默认值 1。因此，对于 guided 调度策略来说，循环分块的方法取决于线程个数（ N ）、迭代次数（ β_0 ）和块大小（ S ）。

例如，给定一个循环，β_0 =800，N =2，S =80，循环划分为{200, 150, 113, 85, 80, 80, 80, 12}。当 π_4 小于 80 时，它就取值为 80。当剩余未调度迭代个数小于 S 时，最后一个块大小的上界会根据需要进行调整。英特尔 C++和 Fortran 编译器所支持的 guided 调度策略是遵从 OpenMP 2.5 标准来实现的。

通过使用 dynamic 和 guided 调度机制，开发人员可以对应用程序进行调整，以应对循环的各个迭代工作量不统一或某些处理核（或处理器）比其它的核（或处理器）运行的快的情形。在一般的情况下，guided 调度策略比 dynamic 调度策略的性能好，因为它的开销要少一些。

runtime 调度方案本质上不是一种调度方案。如果在 schedule 子句中指定 runtime 作为调度策略，OpenMP 运行时环境就对当前的 for 循环使用由 OMP_SCHEDULE 环境变量所指定的调度方案。OMP_SCHEDULE 环境变量的格式是 schedule-type [, chunk-size]。例如：

```
export OMP_SCHEDULE=dynamic, 16
```

使用 runtime 调度策略可以为终端用户提供一定的灵活性，使其能够通过使用 OMP_SCHEDULE 环境变量在前面所提及的三种调度机制中进行动态选择。OMP_SCHEDULE 的默认值是 static。

此外，了解循环调度与分块方案将极大的帮助程序员选择正确的调度策略，有助于避免应用程序在运行时出现的伪共享（false-sharing），从而实现较好的负载平衡。考虑如下示例：

```
float x[1000], y[1000];
#pragma omp parallel for schedule(dynamic, 8)
for (k = 0; k < 1000; k++){
    x[k] = cos(k) * x[k] + sin(k) * y[k];
}
```

假设有一个双核处理器系统，其 cache 线大小是 64 字节。对于上面给出的示范代码，在一个 cache 线中可以放两个迭代块（或两个部分数组），因为块大小在 schedule 子句中被设置为 8。这样的话，数组 x 的每个迭代块都用去 cache 线中的 32 字节，两个迭代块数据可以放在同一个 cache 线中。因为这两个块有可能被两个线程同时读写，所以即使两个线程不会读/写同一数据，也可能会导致一些 cache 线被作废。这就称为伪共享，也就是说实际上没

有必要在两个线程间共享这个 cache 线。有一种比较简单的解决方法，就是使用 schedule(dynamic, 16)，这样一个块就能占据整个 cache 线，从而消除伪共享。通过使用与 cache 线大小相适应的块大小设置来消除伪共享可以极大的改善应用程序的性能

5.3 OpenMP 多线程应用程序性能分析

使用 OpenMP 技术的最主要的目的就是通过多线程的方式能够提高应用程序的执行效率。实际上执行效率是由多个方面的因素共同影响的，在程序编写完成之后，在执行效率达不到预期目标的时候，需要仔细检查，考虑执行期可能产生影响的因素，如果有必要，还可能需要重新改编算法，重写程序才能够获得满意的执行效率。本小节将讲述在优化程序的过程中可能需要考虑的一些因素并在实际工作的过程中采用一些方法来优化程序。

5.3.1 影响性能的主要因素

在考虑影响性能的因素的时候，我们不得不提到著名的 Amdahl 定律。Amdahl 定律是一个关于加速比的定律。加速比的含义是通过并行化之后，程序的执行速度与未并行化之前的执行速度的比值。考虑 F 是程序中并行部分所占用的比率，而 S_p 是并行部分所能够获得的加速比。则程序总体的加速比 S 由下面的公式给出（定律的推导不在这里给出）：

$$S = \frac{1}{(1-F) + \frac{F}{S_p}}$$

从上面的公式可以看出，程序并行部分的比率非常重要。如果并行部分的比率非常小，比如接近于 0，则并行部分的加速比再大对结果的影响也是非常小的。因此，根据阿姆达尔定律，我们应当尽量提高程序中并行部分的比率。为了提高并行化代码在应用程序中的比率，一般需要根据应用程序的特点来确定，对于不同的应用程序有不同的并行化方法，有时需要采用特殊的算法才能够获得令人满意的并行化代码比率。

除此之外，在编写 OpenMP 应用程序的时候，我们一般还需要考虑其它的一些因素引起的效率的降低。实际上很难把程序优化的方法讲解得比较完整，在这里，我们根据 OpenMP 编程的特点，给出在编写 OpenMP 程序时需要考虑的程序优化的一些方面的问题。

OpenMP 本身的开销

OpenMP 获得应用程序多线程并行化的能力不是凭空而来的，而是需要一定的程序库的支持。在这些运行时库对程序并行加速的同时需要运行库的本身，因此，库中代码的运行必然会带来一定的开销。实际上，并不是所有的代码都是需要并行化的，非常有可能在某些情况下，并行化之后的运行效率反而比不上串行执行的效率。这里面有很大一部分原因是由于使用 OpenMP 进行并行化之后 OpenMP 本身引入的开销。因此，只有并行执行代码段负担足够大，而引入的 OpenMP 本身的开销又足够小，此时引入并行化操作才能够加速程序的执行。

负载均衡

我们知道，一个 OpenMP 应用程序在执行的过程中，有很多的同步点，线程只有在同步点进行同步之后才能够继续执行下面的代码。因此，某一个线程在执行到同步点的时候若没有进一步的工作需要完成，此线程只有等待其它线程执行完毕才能够继续。此时，如果各个线程之间的负载不均衡，就有可能造成某些线程在执行过程中无事可干，经常处于空闲状态；而另外一些线程则负担沉重，需要很长时间才能够完成任务。这就是负载均衡给程序并行化带来的新的问题。在使用 OpenMP 进行并行程序编码的时候，要非常注意使得线程之间的负载大致均衡，能够让多个线程在大致相同的时间内完成工作，从而能够提高程序运行的效率。OpenMP 的运行时库以及环境变量对负载均衡有一定的支持，例如可以划分执行的粒度，并通过动态调度的方法消除一定的负载均衡问题。

局部性

现代计算机的中央处理单元都拥有一定数量的高速缓存（cache）。高速缓存距离中央处理单元非常近，通常情况下处于一块芯片之内。由于有这些结构上的特点，高速缓存的访问速度非常快，一般要比访问内存速度快上一个数量级。同时，高速缓存存在于中央处理单元的内部，其容量是非常有限的，基本上不可能将整个应用程序装入到高速缓存中。在实际的运行过程中，高速缓存将缓存最近刚刚访问过的数据以及代码以及这些数据与代码相邻的数据与代码。这样的行为是基于局部性的假设：即程序在继续执行的过程中，继续访问同样的或者相邻的数据的可能性要比随机访问其它数据的可能性要大。因此，在编写程序的时候，需要考虑到高速缓存的作用，有意地运用这种局部性带来的高速缓存的效率提高。

线程同步带来的开销

对于多线程应用程序来说，相对于串行执行程序的一个固有特点就是线程之间可能存在的同步开销。多个线程在进行同步的时候必然带来一定的同步开销。当然，有的同步开销是不可避免的，但是很多情况下，不合适的同步机制或者算法会带来运行效率的急剧下降。在使用多线程进行应用程序开发的时候对这一点需要非常的注意，考虑同步的必要性，消除不必要的同步，或者调整同步的顺序，就有可能带来性能上的提升。

5.3.2 OpenMP 程序性能分析实例

在这一个小节中，我们将通过例子来说明上述的性能影响因素，以及可能的优化实例。为了能够明确地看到优化的效果，需要通过某种方式来衡量程序执行的效率。为了得到程序的性能指标，可以通过衡量时间的方法获得。一般的时间函数粒度太大，毫秒级的精度不足以满足小程序运行效率的需求。一般可以通过读取处理器寄存器的方法得到周期精确的时间度量，假定我们使用的处理器为英特尔 IA32 架构，则我们可以读取 rdtsc 寄存器来精确度量程序的全部、部分执行时间，如下所示。如为其它处理器架构，则请查阅相关手册。

代码 5.25 如何通过软件的方法得到 x86 测试平台的主频

```
#include <unistd.h>
#include <sys/times.h>

/* Keep track of most recent reading of cycle counter */
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;
```

```
void access_counter(unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}

void start_counter()
{
  access_counter(&cyc_hi, &cyc_lo);
}

double get_counter()
{
  unsigned ncyc_hi, ncyc_lo;
  unsigned hi, lo, borrow;
  double result;
  /* Get cycle counter */
  access_counter(&ncyc_hi, &ncyc_lo);
  /* Do double precision subtraction */
  lo = ncyc_lo - cyc_lo;
  borrow = lo > ncyc_lo;
  hi = ncyc_hi - cyc_hi - borrow;
  result = (double) hi * (1 << 30) * 4 + lo;
  if (result < 0) {
    fprintf(stderr, "Error: Cycle counter returning negative value: %.0f\n",
result);
  }
  return result;
}


/* Determine clock rate by measuring cycles
   elapsed while sleeping for sleeptime seconds */
double mhz_full(int verbose, int sleeptime)
{
  double rate;
  start_counter();
  sleep(sleeptime);
  rate = get_counter()/(1e6*sleeptime);
  if (verbose)
```

```
    printf("Processor Clock Rate ~= %.1f MHz\n", rate);
  return rate;
}

/* Version using a default sleeptime */
double mhz(int verbose)
{
  return mhz_full(verbose, 2);
}

int main (int argc, char * argv[])
{
        double frequency;

        frequency = mhz(0);
        printf(“frequency = %f MHz\n”, freqency);
        return 0;
}
```

上述程序可以用来得到所测试平台的主频。在每一个需要精确测量时间的函数或程序段的前后加上 start_counter()和 get_counter()函数对就可以以周期精确的精度得到所需要测量的函数或程序段的执行周期数，再除以测试平台的系统主频，就可以得到一个较精确的时间度量。下面是上述程序的一次运行结果，(运行的平台是 2 路 4 核英特尔® Xeon® E5310 @1.60GHz， 2G 的内存，下面的所有测试都基于这个平台，如非特别指出，设置 OMP_NUM_THREADS=2)：

```
frequency = 1597.0 MHz
```

而如果我们使用

```
$ cat /proc/cpuinfo | grep MHz
```

来得到系统主频的话，可以得到如下显示：

```
cpu MHz         : 1596.078
```

可见程序测量与系统实际主频是相当接近的。

并行化带来的额外负担

前面已经说过，并行化会带来额外的负担，因此，从效率上考虑，并不是所有的程序都应当并行化的。特别是对于小程序来说，并行化带来的效率不足以弥补并行化本身带来的运行负担,勉强进行并行化就会得不偿失。下面的程序就演示了在小循环并行化上产生的额外负担：

代码 5.26 小循环并行化上产生的额外负担

```
int main (int argc, char * argv[])
{
double diff;
unsigned long sum = 0;

        start_counter();
#pragma omp parallel for reduction (+:sum)
        for (int i=0;i<10000;i++)
            sum+=i;
        diff = get_counter();

        printf ("sum=%ul execution with OpenMP, count=%f\n",sum,diff) ;

        sum=0;
start_counter();
        for (int i=0;i<10000;i++)
            sum+=i;
        diff = get_counter();

        printf ("sum=%ul serial execution, count=%f\n",sum, diff) ;
}
```

程序是一个简单的求和操作，从 1 求和到 10000，第一个循环使用了 OpenMP 对循环进行并行化，而第二个循环则使用了简单的串行执行方式。下面是程序的一次执行结果：

```
sum=499950001 execution with OpenMP, count=520452.000000
sum=499950001 serial execution, count        =102.000000
```

可以看到，串行执行的效率要比并行执行的效率高一个数量级以上，这主要是由于循环的规模比较小，使用并行化带来的效果无法抵消并行化的额外负担。但是，如果将上述的 10000 改成 100000000，则循环的执行结果就变成：

```
sum=887459712l execution with OpenMP, count=78614544.000000
sum=887459712l serial execution, count       =100159806.000000
```

此时，并行化带来的效率提高大大超过了相应的负担，基本上两个核心都能够得到独立的计算。从这样一个简单的例子我们就可以明显地看到在编写并行化程序的时候，应当尽量使得程序真正工作的负载超过并行化的负担，每一个线程负担的工作要足够多，这样才能够获得并行化之后的效率提高。

负载均衡

通过一个简单的程序我们就可以明显看出负载均衡对程序性能的影响。下面的程序有两个函数，分别具有不同的负担，轻负担的函数实际上就是一个空函数，而重负担的函数则用来求和（从 1 到 100000000）。

代码 5.27 负载均衡

```
#include <omp.h>

void smallwork()
{

}

void bigwork()
{
        unsigned long sum=0;

        for (int i=0;i<100000000;i++)
            sum+=i;
}

int main (int argc, char * argv[])
{
double diff;
        start_counter();
#pragma omp parallel for
        for (int i=0;i<100;i++) {
            if (i<50)
                smallwork () ;
            else
                bigwork () ;
        }
diff = get_counter();

        printf ("count=%f\n",diff) ;

        start_counter();
#pragma omp parallel for
        for (int i=0;i<100;i++) {
            if (i%2)
                smallwork () ;
            else
                bigwork () ;
        }
diff = get_counter();
```

```
        printf ("count=%f\n",diff) ;
    }
```

下面是程序的一次运行结果：

```
count=780984.000000
count=106848.000000
```

可以看到，两个循环的工作量是一样的，但是运行时间几乎相差了 7 倍。在第一个循环中，由于步长是 1，OpenMP 运行时环境将前面 50 个循环分配个 0 号线程，将后面 50 个循环分配给 1 号线程。1 号线程需要运行的都是负担沉重的函数，而 0 号线程会很快执行完 50 个空函数，继续等待 1 号线程完成工作。在第二个循环中，OpenMP 运行时环境仍然将前面 50 个循环分配个 0 号线程，将后面 50 个循环分配给 1 号线程，但是，负载的分配发生很大的变化，轻重负载被均衡地分配给两个线程，因此两个线程会在相当的时间内完成工作。通过负载均衡的办法，我们就获得了执行效率的提高。

在循环并行化中，由于循环的次数在进入循环并行化的时候已经确定了，因此，在具体的 OpenMP 库与运行时实现的时候，一般的做法是将循环次数平均分配到所有的线程当中。当然，有的时候这会引起一些性能的损失。在每一次循环的工作负担大致相当的情况下，这种按照循环次数平均的策略可能会获得最大的负载平衡，即各个线程完成的时间大致相当。但是，实际的程序可能每一次的循环工作负载不太一样，会根据循环变量的不同，工作负载会有一个很大的波动。此时，这种通过静态平均的策略进行线程的调度就会引起负载的不均衡，有可能造成某些线程由于负载比较轻，早早的完成工作，而其它线程由于负载非常重，需要很长的时间才能完成工作。因此，OpenMP 除了静态调度策略外，支持了动态的调度策略。可以根据工作负载的轻重，将循环次数分成几个循环块，每一次一个线程只执行一个循环块。当某一个循环块执行结束的时候，通过调度，此线程可以获得下一个循环块。通过这种动态的方式可以获得负载平衡。因此，针对不同的应用，OpenMP 可以通过 static，dynamic，guided，runtime 等子句来知道调度器采用合适的调度策略。环境变量 OMP_SCHEDULE 用来配置 runtime 类型的调度策略，即根据环境变量，runtime 子句可以采用不同调度策略。例如当 OMP_SCHEDULE 设置为“dynamic,3”的时候，说明程序希望采用的调度策略是动态调度策略，并且循环块的次数为 3 次。具体操作方法可以参考各个平台的不同 OpenMP 实现。

同步开销

在某些情况下，使用同步不当，会造成性能的损失。此时，需要考虑一些新的算法或者构造新的程序执行流程，以便降低同步带来的开销。

代码 5.28 同步开销

```
#include <omp.h>

int main(int argc, char * argv[])
{
double diff;
```

```
    int i;
    unsigned long sum = 0;

            start_counter();
            for (i=0;i<10000000;i++)
                sum+=i;
            diff = get_counter();
    printf("sum=%ul serial execution count=%f\n",sum,diff) ;

            sum=0;
            start_counter();
    #pragma omp parallel for
            for (i=0;i<10000000;i++)
    #pragma omp critical
                sum+=i;
            diff=get_counter();
            printf("sum=%ul parallel with critical count=%f\n",sum,diff);

            sum=0;
    start_counter();
    #pragma omp parallel for reduction (+:sum)
            for (i=0;i<10000000;i++)
                sum+=i;
            diff = get_counter();
            printf ("sum=%ul parallel with reduction count=%f\n",sum,diff);
    }
```

上述程序的一次运行结果：

```
sum=22807072641 serial execution count=10007286.000000
sum=22807072641 parallel with critical count=5052602004.000000
sum=22807072641 parallel with reduction count=5302530.000000
```

在上述程序中，第一个循环是串行执行的；第二个循环是在第一个循环的基础上加上了并行化的支持，但是为了消除数据冲突，将 sum+=1 的操作定义为了临界区。因此该指令在执行时将增加加锁、开锁的开销。虽然结果正确，但是由于过多的加锁、开锁的开销，负担沉重，实际产生的程序运行行为是串行的，运行效率低下；第三个循环才是在 OpenMP 环境中正确使用的方式，既利用了多线程并行执行的效率提高，又避免了同步带来的额外开销。